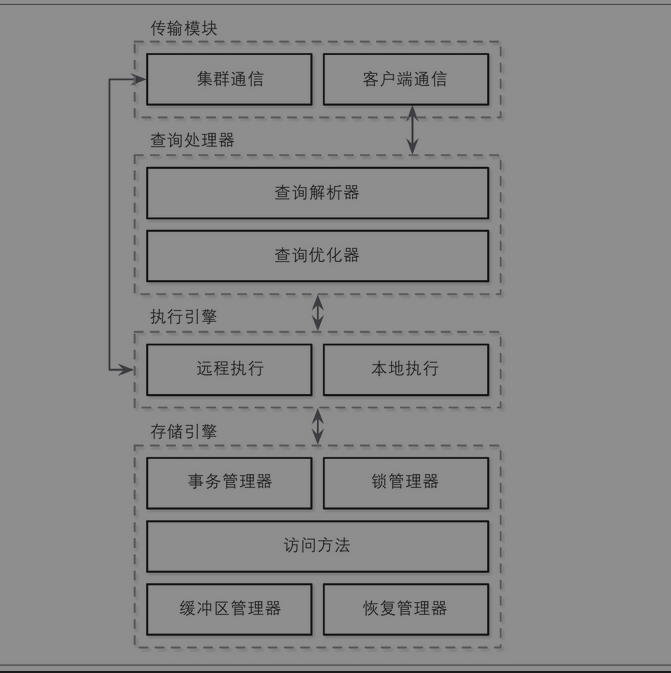
自己动手写数据库

数据库架构



‌**关键点在于理解数据存储原理(涉及很多数据结构知识)、查询优化机制和事务管理逻辑**‌。

* 基础表操作（建表、删表、IUD操作）
* 索引支持（如B树索引加速查询）
* 查询解析与优化（SELECT语句、ORDER BY排序）
* 事务管理（日志系统、锁机制、崩溃恢复）
* 高级特性（Buffer Pool缓存、Catalog表管理）

目录

1. 引言
2. 从文件系统到数据库
3. 索引数据结构
4. B树与故障恢复
5. B+树节点结构与插入操作
6. B+树删除操作与测试
7. 仅追加键值存储
8. 空闲列表：回收与复用
9. 基于KV的表存储
10. 范围查询
11. 二级索引
12. 原子事务
13. 并发控制
14. SQL解析器
15. 查询语言

**引言**

**通过自建数据库掌握核心原理‌**

‌**学习路径指南**‌  
复杂的数据库系统是基于若干基础原则构建的：

1. ‌**原子性与持久性**‌
   * 数据库不仅是文件存储
   * 通过fsync实现数据持久化
   * 崩溃恢复机制
2. ‌**基于B树的键值存储**‌
   * 磁盘数据结构设计
   * 使用空闲列表管理存储空间
3. ‌**在KV存储上构建关系型数据库‌**
   * 表与索引到底层B树的映射逻辑
   * 类SQL查询语言的解析器与解释器实现
4. ‌**事务的并发控制**‌

**‌用3000行代码逐步实现数据库‌**

令人惊叹的是，如此广泛的技术主题可通过3000行代码完整呈现。

* 虽然您可能接触过更大规模的项目，但代码行数≠技术深度

**关键开发阶段与技术映射**‌

1. ‌**366行**‌ - B+树基础结构  
   • 实现磁盘友好的平衡树结构，支持有序检索  
   • 采用16KB块大小优化磁盘I/O
2. ‌**601行**‌ - 仅追加(append-only)键值存储  
   • 通过WAL日志实现崩溃恢复  
   • 数据不可变性确保原子操作
3. ‌**731行**‌ - 实用KV存储（含空闲列表）  
   • 空间回收机制管理磁盘碎片  
   • 支持页面压缩降低存储开销
4. ‌**1107行**‌ - 基于KV的关系表  
   • B+树叶子节点映射表数据  
   • 行存储格式设计（如128字节/行）
5. ‌**1294行**‌ - 范围查询  
   • 利用B+树有序特性实现高效扫描
6. ‌**1438行**‌ - 二级索引  
   • 多棵B+树协同工作  
   • 索引列与主键关联机制
7. ‌**1461行**‌ - 事务接口  
   • 支持ACID特性  
   • 基于锁的冲突检测
8. ‌**2795行**‌ - 类SQL查询语言  
   • 解析器生成查询执行计划  
   • 优化器选择索引访问路径

**实践出真知：掌握原理而非术语**

数据库领域充斥着含义模糊、定义混乱的专业术语，阅读文献时极易迷失方向。正如费曼所言："无法亲手构建的东西，就不能算真正理解"。仅靠阅读数据库理论就能构建数据库吗？不妨用实践检验你的理解！

虽然知识体系庞大，但核心原理才是关键——构建数据库只需掌握少数基本原则，每个人都可以尝试。

**主题1：持久性与原子性**

**超越数据格式的保障**

智能手机广泛使用SQLite（基于文件的数据库）而非JSON等格式存储数据，核心原因在于‌**崩溃安全**‌：

* 直接操作文件时，系统崩溃可能导致：  
  • 数据半写入（部分更新）  
  • 文件截断  
  • 文件丢失
* SQLite通过‌**预写日志(WAL)**‌实现原子更新

**通过fsync实现持久性与原子性**

1. ‌**原子性**‌
   * 确保操作要么完整执行，要么完全不生效（无中间状态）
   * 例：银行转账需同时更新双方账户，不能仅完成单边操作
2. ‌**持久性**‌
   * 数据一旦提交即永久存储，不受系统崩溃影响
   * 需克服存储层级缓存（OS页缓存/设备RAM）的延迟写入问题
3. ‌**技术实现**‌
   * fsync系统调用强制刷盘并等待完成，解决持久性问题
   * 结合‌**日志校验和**‌与‌**双写机制**‌保障原子性

（注：SQLite的设计融合了事务日志与存储引擎协同，比裸文件操作可靠性提升2个数量级）

**主题2：索引数据结构**

**通过索引控制延迟与成本**

数据库将查询隐式转换为结果，但性能优化需同时考虑：

1. ‌**结果准确性**‌
2. ‌**延迟敏感度**‌（响应时间）
3. ‌**资源成本**‌（内存/I/O/计算）

**OLAP与OLTP的索引策略差异**

| **类型** | **数据规模** | **操作特征** | **索引策略** | **性能目标** |
| --- | --- | --- | --- | --- |
| OLAP | 大数据量 | 聚合/连接操作 | 有限索引或无索引1 | 吞吐量优先 |
| OLTP | 小数据量 | 点查询/简单范围查询 | 密集索引37 | 低延迟&低成本1 |

（注："事务型(OLTP)"术语与数据库事务无关，仅为行业特定表述1）

**索引技术的核心价值**

* ‌**B+树**‌：OLTP场景标准方案，平衡读写效率
* ‌**哈希索引**‌：适用于KV存储，但不适用范围查询
* ‌**学习型索引**‌：减少内存占用，适配异构存储
* ‌**混合索引**‌：如IVF+HNSW组合优化向量检索

内存数据结构 vs 磁盘数据结构  
将索引数据结构放在磁盘上会面临额外挑战, 其中一个问题是原地更新磁盘数据，因为必须处理崩溃后的损坏状态。磁盘不仅仅是速度较慢的内存。  
RAM中的"R"代表"随机"，这对基于磁盘的数据来说又是个问题，因为随机访问比顺序访问慢得多，即使在SSD上也是如此。因此像二叉树这样的结构不可行，而B树和LSM树则适用。数据结构的并发访问也是需要讨论的话题。

**主题3：基于KV的关系型数据库**

**数据库的两层接口结构**

1. ‌**SQL层**‌（用户接口）

* 虽常被等同于数据库，但本质是交互界面
* 并非数据库核心，底层功能实现才是关键

1. ‌**KV层**‌（存储引擎接口）

* 基础操作：get/set/delete单键值
* 核心能力：‌**有序范围遍历**‌（range scan）
* 关系型数据库实际构建在类KV的存储引擎之上

**查询语言实现逻辑**

* ‌**解析器与解释器**‌
  + 代码量虽大但实现简单
  + 仅需递归即可完成构建（适用于任何计算机语言）
  + 该原理可延伸至：  
    ✓ 现有语言解析  
    ✓ 自研编程语言/DSL开发  
    （进阶参考著作《从源代码到机器码》）

（注：KV接口的简洁性使其成为构建高层数据库的理想基础层）

从文件系统到数据库

让我们从文件系统开始，深入探讨其面临的挑战。

1.1 原地更新文件

假设您需要将某些数据持久化到磁盘，以下是典型的实现方案

//java code

该代码实现了以下核心功能与缺陷：

一、核心机制

1. 文件操作特性：

* 当目标文件不存在时自动创建新文件
* 对已存在文件执行清空后写入（truncate-and-write）
* 必须显式调用fsync(fp.Sync())才能确保数据持久化

二、主要技术缺陷

1. 全量更新问题：

* 仅适合微小型数据存储
* 类比场景：这正是Excel不能作为数据库使用的根本原因

1. 崩溃风险：

* 修改流程需"读取内存→修改→全量覆写"
* 覆写过程中发生崩溃会导致原始数据损毁

1. 并发控制缺失：

* 读取端可能获取到部分更新的混乱数据
* 多写入端会产生操作冲突
* 主流数据库采用客户端-服务端架构的核心原因正在于此
* 无服务端协调时并发控制极其复杂（如SQLite的实现方式）

1.2 原子化重命名  
通过文件重命名实现数据原子替换

采用非就地更新（non-in-place update）策略可解决诸多问题：先写入新文件，再删除旧文件。

保留旧文件数据的优势在于：

1. 更新中断时可回退：旧文件保持完整，确保数据可恢复性
2. 并发读取安全性：读操作不会获取到部分写入的脏数据

核心挑战在于新文件的发现机制，通用解决方案是将新文件重命名为旧文件路径

文件重命名替换现有文件时，该操作是原子性的——无需（也不应该）先删除旧文件。

需特别注意技术术语的精确含义：当看到"X是原子操作"时，必须追问"X在什么维度上是原子的？"。本例中：

* 重命名操作（rename()）对并发读取者具有原子性：
  + 读取者要么访问完整的旧文件
  + 要么访问完整的新文件35
  + 不会出现部分更新或中间状态15
* ‌**非原子性风险**‌：
  + 重命名操作可能因系统崩溃而丢失（仅写入内存缓存，未持久化到磁盘）25
  + 需显式调用fsync()确保元数据写入存储设备

为什么重命名会起作用？  
文件系统会将文件名与文件数据进行映射，因此通过重命名来替换文件只是将文件名指向新的数据，而不会触及旧数据。这就是为什么文件系统中可以实现原子重命名。而且，无论数据大小如何，操作成本都是恒定的。  
在 Linux 系统中，如果旧文件仍被某个读取器打开，则重命名替换后该旧文件可能仍然存在；只是无法通过文件名访问。读取器可以安全地处理其获取的任何版本的数据，而写入器不会被读取器阻塞。然而，必须有一种方法来防止并发写入。并发级别为多读单写，这就是我们要实现的。

1.3 只追加日志  
使用日志实现安全的增量更新  
实现增量更新的一种方法是将更新内容追加到文件中。之所以称为“日志”，是因为它是只追加的。与就地更新相比，它更安全，因为不会覆盖任何数据；即使发生崩溃，也可以恢复旧数据。  
使用日志时，读取器必须考虑所有日志条目。例如，这里有一个基于日志的键值对存储，包含 4 条日志记录：  
0 1 2 3  
| 设置 a=1 | 设置 b=2 | 设置 a=3 | 删除 b |  
最终状态为 a=3。  
日志是许多数据库的重要组成部分。然而，日志只是对每次更新的描述，这意味着：  
• 它不是索引数据结构；读取器必须读取所有条目。  
• 它无法从已删除的数据中回收空间。  
因此，仅靠日志不足以构建数据库，必须与其他索引数据结构结合使用。

带有校验和的原子日志更新  
虽然日志不会损坏旧数据，但在崩溃后，如果最后一个条目被损坏，您仍然需要处理它。有很多可能性：1. 最后一次追加操作根本就没有执行；日志仍然是好的。2. 最后一条记录只写了一半。3. 日志的大小增加了，但最后一条记录却不在那里。

处理这种情况的方法是在每个日志条目中添加一个校验和。如果校验和错误，说明更新没有发生，从而使日志更新对于读取者和持久性而言都是原子性的。  
此场景涉及在成功执行 fsync 之前发生的不完整写入（在数据库术语中称为“撕裂写入”）。校验和还可以检测 fsync 之后出现的其他形式的损坏，但这不是数据库可以恢复的情况。

1.4 `fsync` 使用陷阱

重命名文件或创建新文件后，必须在父目录上调用fsync。A.

目录是从文件名到文件的映射，与文件数据一样，除非你对其进行持久化，否则它不会持久存在

使用fsync。请看这个关于目录上fsync的示例[1]。

fsync的另一个问题是错误处理。如果fsync失败，数据库更新也会失败，但如果是这样呢

你之后阅读了文件吗？即使fsync失败（因为……），你也可能会获取到新数据

操作系统页面缓存）！这种行为依赖于文件系统[2]。

1.5数据库挑战概述

我们学到了什么：

1.就地更新的问题。

•通过重命名文件来避免就地更新。

•避免使用日志进行就地更新。

2.仅附加日志。

•增量更新。

•不是一个完整的解决方案；没有索引和空间重用。

3.fsync用法

有待解决的问题:

1.索引数据结构以及如何更新它们。

2.重复使用仅追加文件的空间。

3.将日志与索引数据结构相结合。

4.并发性。

索引数据结构

2.1查询类型

大多数SQL查询可以分为3种类型：

1.扫描整个数据集。（未使用索引）。

2.点查询：按特定键查询索引。

3.范围查询：按范围查询索引。（索引已排序）。

有一些方法可以使扫描更快，例如基于列的存储。但扫描是𝑂（𝑁）

无论它有多快；我们的重点是使用数据在𝑂（log𝑁）中提供的查询

结构。

范围查询由两个阶段组成：

1.查找：找到启动键。

2.迭代：按排序顺序查找上一个/下一个键。

点查询只是查找，没有迭代；我们只需要一个排序数据结构

2.2哈希表

如果只考虑点查询（get、set、del），哈希表是可行的，因此我们不会费心

因为缺乏秩序。

然而，对哈希表进行编码，即使是在内存中，也是一项有价值的练习。离这儿很远

与B树相比，我们将在后面进行编码更容易，尽管仍然存在一些挑战：

•如何生成哈希表？当加载时，键必须移动到更大的哈希表中

系数太高。一次移动所有内容是禁止的O（N）。重新整理必须

逐步完成，即使是内存中的应用程序，如Redis。

•前面提到的其他事项：就地更新、空间重用等。

2.3排序阵列

排除哈希表，让我们从最简单的排序数据结构开始：排序数组。

您可以在𝑂（log𝑁）中对其进行二进制搜索。对于字符串（KV）等可变长度数据，请使用

进行二进制搜索的指针（偏移量）数组。

更新排序数组是𝑂 (𝑁)，要么到位要么不到位。所以这不实用，但可以

扩展到其他可更新的数据结构。

降低更新成本的一种方法是将阵列拆分为几个较小的不重叠阵列

数组-嵌套的排序数组。此扩展导致B+树（多级n元树），其中

维护这些小阵列（树节点）的额外挑战。

“可更新阵列”的另一种形式是日志结构合并树（LSM树）。更新

首先在较小的数组（或其他排序数据结构）中缓冲，然后合并到

主数组变得太大时。更新成本通过传播较小的

数组转换为更大的数组。

2.4 B树

B树是平衡的n叉树，与平衡的二叉树相当。每个节点存储

键（和分支）的数量可变，最多可达𝑛 and 𝑛 > 2。

使用较短的树减少随机访问

磁盘每秒只能执行有限数量的IOs（IOPS），这是限制

树查找的因子。树的每个级别都是在查找中读取的磁盘，n元树是

对于相同数量的密钥，短于二叉树（log与log2），因此n元树

用于每次查找较少的磁盘读取。

如何选择？有一个权衡：

•更大的𝑛意味着每次查找的磁盘读取更少（更好的延迟和吞吐量）。

•较大的𝑛意味着较大的节点，更新较慢（稍后讨论）。

以页面为单位的IO

虽然您可以在文件的任何偏移量处读取任意数量的字节，但磁盘不能这样工作

方式。磁盘IO的基本单位不是字节，而是扇区，它们是512字节连续的

旧HDD上的块。

然而，磁盘扇区不是应用程序关心的问题，因为常规文件IO不会

直接与磁盘交互。操作系统在页面缓存中缓存/缓冲磁盘读/写，

它由称为页面的4K字节内存块组成。

无论如何，都有一个最小的IO单位。DB还可以定义自己的IO单位（也可以

称为页面），它可以大于操作系统页面。

最小IO单元意味着树节点应以单元的倍数分配；

一半使用的单元是一半浪费的IO。另一个反对小𝑛的原因！

B+树变体

在数据库上下文中，B-tree是B-tree的变体，称为B+树。在B+树中，

内部节点不存储值，值仅存在于叶节点中。这导致树变短

因为内部节点具有更多的分支空间。

B+树作为内存中的数据结构也是有意义的，因为最小IO单元

RAM和CPU缓存之间为64字节（缓存线）。性能优势不是

就像在磁盘上一样棒，因为64个字节中容纳不了多少。

数据结构空间开销

二叉树不切实际的另一个原因是指针的数量；每个键具有

至少1个来自父节点的传入指针，而在B+树中，叶中有多个键

节点共享1个传入指针。

叶节点中的键也可以以紧凑格式打包或压缩以进一步减少

空间。

2.5日志结构化存储

通过合并更新：摊销成本

日志结构存储最常见的例子是日志结构合并树（LSM树）。

它的主要思想既不是日志也不是树；而是“合并”！

让我们从两个文件开始：一个小文件保存最近的更新，一个大文件保存其余的更新

数据的。更新首先转到小文件，但它不能永远增长；它将被合并

当它达到阈值时放入大文件中。

写入=>|新更新|=>|累积数据|

文件1文件2

合并2个排序的文件会产生一个更新、更大的文件，该文件将替换旧的大文件并缩小

小文件。

合并为𝑂（𝑁），但可以与读者和作者同时进行。

减少多个级别的写入放大

缓冲更新比每次重写整个数据集要好。如果我们延长

这个方案是多层次的吗？

在2级方案中，每当小文件达到阈值时，都会重写大文件，

多余的磁盘写入称为写入放大，并且随着大文件的变大，情况变得更糟。

如果我们使用更多的级别，我们可以通过将第二个级别合并到第三个级别来保持其较小，

类似于我们如何保持第一级小。

直观地说，级别呈指数级增长，两个增长的力量（类似地合并

大小级别）导致写入放大最少。但在写入之间存在权衡

放大和级别数（查询性能）。

LSM树索引

每个级别都包含索引数据结构，它可以只是一个排序的数组，因为

级别从不更新（第一级除外）。但二进制搜索并不比

就随机访问而言的二叉树，因此明智的选择是在级别内使用B树，

这是LSM树的“树”部分。无论如何，数据结构要简单得多，因为

缺乏更新。

为了更好地理解“合并”的思想，您可以尝试将其应用于哈希表，即。

日志结构哈希表。

LSM树查询

键可以在任何级别中，因此要查询LSM树，需要组合每个级别的结果

（用于范围查询的n向合并）。

对于点查询，Bloom过滤器可以用作优化，以减少

搜索的级别。

由于级别从不更新，因此在较旧的级别中可以存在旧版本的键，并将其删除

在较新的级别（称为逻辑删除）中，键用特殊的标志进行标记。因此，更新的级别

在查询中具有优先级。

合并过程自然地从旧的或已删除的键中回收空间。因此，它也被称为

压实度。

真实LSM树：SSTable、MemTable和log

这些是关于LSM树实现细节的行话。你不需要认识他们

从原则上建立一个，但它们确实解决了一些实际问题。

级别被拆分为多个称为SSTables的非重叠文件，而不是一个大型文件

文件，以便可以逐步进行合并。这减少了以下情况下的可用空间要求：

合并大型级别，并且合并过程随时间展开。

第一级直接更新，日志成为可行的选择，因为第一级是

在大小上有界限。这是LSM树的“日志”部分，是组合日志的示例

使用其他索引数据结构。

但即使日志很小，仍然需要适当的索引数据结构。日志数据

在名为MemTable的内存索引中重复，该索引可以是B-tree、skiplist或

不管怎样。它是少量有限的内存中数据，并且具有以下附加好处

加速读取最近的更新场景。

2.6索引数据结构概述

有两个选项：B+树和LSM树。

LSM树解决了上一章中的许多挑战，例如如何更新磁盘-

基于数据结构和结果空间。尽管B+树仍然面临这些挑战

稍后将进行探索。

B树和崩溃恢复

3.1 B树作为平衡n叉树

高度平衡树

许多实用的二叉树，如AVL树[1]或RB树，被称为高度平衡树

树，表示树的高度（从根到叶）限制为𝑂 (log 𝑁)，所以a

查找是𝑂 (log 𝑁)

B树也是高度平衡的；所有叶节点的高度都相同。

泛化二叉树

n元树可以从二叉树中推广（反之亦然）。2-3-4就是一个例子

树，这是一个B树，其中每个节点可以有2个、3个或4个子节点。2-3-4树

相当于RB树。然而，我们不会深入细节，因为它们不是

理解B树所必需的。

可视化排序序列的2级B+树[1，2，3，4，6，9，11，12]。

[1、4、9]

/|个\

v v v v

[1，2，3][4，6][9，11，12]

在B+树中，只有叶节点包含值，键在内部节点中重复以指示

子树的键范围。在本例中，节点[1、4、9]表示其3个子树为

间隔[1，4），[4，9）和[9，+∞）内。然而，3个间隔只需要2个键，

因此，可以省略第一个键（1），并且3个间隔变为（-∞，4）、[4,9）和（9，+∞）。

3.3维护B+树

更新B+树时要保留3个不变量：

1.所有叶节点的高度相同。

2.节点大小受常量限制。

3.节点不为空。

通过拆分节点来生长B树

通过插入叶节点来违反第二个不变量，叶节点通过拆分

节点划分为较小的节点。

父/母

/|\=>/||\

L1 L2 L6 L1 L3 L4 L6级

\*\*（\*）\*

拆分叶节点后，其父节点将获得新的分支，该分支也可能超过

大小限制，因此它可能也需要拆分。节点分裂可以传播到根节点，

将高度增加1。

新根（\_R）

/\

根部N1 N2

/|\=>/||\

L1 L2 L6 L1 L3 L4 L6级

这保留了第一个不变量，因为所有树叶同时增加高度1。

通过合并节点收缩B树

删除可能会导致节点为空。通过合并空节点来恢复第三个不变量

到同级节点中。合并是拆分的反义词。它也可以传播到根

节点，因此树高可以降低。

当编码B树时，可以更早地进行合并以减少浪费的空间：您可以合并

非空节点，当其大小达到下限时。

3.4磁盘上的B-Tree

您已经可以使用这些原则来编写内存中的B树。但磁盘上的B树

需要额外的考虑

基于块的分配

缺少的一个细节是如何限制节点大小。对于内存中的B+树，可以限制

节点中的最大键数，节点大小（以字节为单位）不是问题，因为您

可以根据需要分配任意多个字节。

对于基于磁盘的数据结构，没有可依赖的malloc/free或垃圾收集器；

空间分配和重用完全取决于我们。

如果所有分配的大小都相同，则可以使用空闲列表来完成空间重用，我们将

稍后实施。目前，所有B树节点的大小都相同。

写入时复制B树以进行安全更新

我们看到了3种抗崩溃的磁盘数据更新方法：重命名文件、日志、LSM树。

教训是不要在更新期间销毁任何旧数据。这种想法可以应用于树：

制作节点的副本，改为修改副本。

插入或删除从叶节点开始；在复制修改后，其

父节点必须更新为指向新节点，这也在其副本上完成。

复制会传播到根节点，从而产生新的树根。

•原始树保持完整，可以从旧根访问。

•新根，将更新的副本一直复制到叶，共享所有其他节点

使用原始树

这是更新叶c的可视化。复制的节点为大写（D，B，c），

而共享子树是小写的（a，e）。

这称为写入时复制数据结构。它也被描述为不可变的，仅追加（不

字面意思），或持久（与耐久性无关）。请注意，数据库行话不会

具有一致的含义。

写入时拷贝B树还存在2个问题：

1.当每次更新后树根发生变化时，如何查找树根？碰撞安全问题

被简化为单个指针更新，我们将在后面解决。

1. 如何重用旧版本中的节点？这就是空闲列表的工作。

写入时拷贝B树的优势

保留旧版本的一个好处是我们免费获得快照隔离。A类

事务从树的一个版本开始，不会看到其他版本的更改。

而崩溃恢复是毫不费力的；只使用最新的旧版本。

另一个是它适合多读取器-单编写器并发模型，以及读取器

不要阻止写入程序。我们稍后将探讨这些。

替代方法：使用双写入进行就地更新

虽然崩溃恢复在写入时复制数据结构中很明显，但它们可能是不需要的

由于高写入放大率。每次更新都会复制整个路径（日志），虽然

大多数就地更新仅接触1个叶节点。

可以使用崩溃恢复进行就地更新，而无需写入时复制：

1.在某处保存整个更新节点的副本。这就像是写入时复制，但

而不复制父节点。

2.fsync保存的副本。（此时可以响应客户端。）

3.实际更新数据结构到位。

4.fsync更新。

崩溃后，数据结构可能会更新一半，但我们并不真正知道。我们做了什么

做的是盲目地应用保存的副本，以便数据结构以更新的状态结束，

无论当前状态如何

保存的更新副本在MySQL术语中称为双写[2]。但如果

双重写入已损坏？它的处理方式与日志相同：校验和。

•如果校验和检测到错误的双写，则忽略它。它在第一次fsync之前，因此

主要数据处于良好的旧状态。

•如果双写入良好，则应用它将始终产生良好的主数据。

一些数据库实际上将双写存储在日志中，称为物理日志[3]。有2个

日志记录类型：逻辑和物理。逻辑日志描述高级操作，如

插入密钥，这样的操作只能在数据库处于良好状态时应用于数据库，因此

只有物理日志记录（低级磁盘页面更新）对恢复有用。

碰撞恢复原则

让我们比较一下双写和写入时复制：

•双重写入使更新幂等；DB可以通过应用

保存的副本，因为它们是完整节点。

•写入时拷贝自动将所有内容切换到新版本。

它们基于不同的想法：

•双重写入确保有足够的信息来生成新版本。

•写入时复制确保保留旧版本。

如果使用双写保存原始节点而不是更新的节点，会怎么样？那就是

从损坏中恢复的第三种方法，它将恢复到旧版本，如复制-

写入。我们可以将这三种方法组合为一个想法：有足够的信息用于

任何时候的旧状态或新状态

此外，始终需要一些复制，因此较大的树节点更新速度较慢。

我们将使用写时复制，因为它更简单，但您可以在这里有所偏离。

3.5我们学到了什么

B+树原则：

•n元树，节点大小受常数限制。

•所有叶片的高度相同。

•拆分和合并以进行插入和删除。

基于磁盘的数据结构：

•写入时复制数据结构。

•双写崩溃恢复。

我们现在可以开始编码了。基于B+树创建持久KV的3个步骤：

1.对B+树数据结构进行编码。

2.将B+树移动到磁盘。

3.添加空闲列表。

B+树节点和插入

**.1 设计 B+树节点**

**实现步骤**

我们将从底层逐步构建 B+树数据结构，其他数据库相关功能将在后续章节补充。具体流程如下：

1. ‌**设计节点格式**‌
   * 定义包含所有必要信息的节点结构（类型、键数量、子指针、KV 对、偏移量列表）。
2. ‌**写时复制（Copy-on-Write）操作节点**‌
   * 通过复制节点的方式实现键的插入（insert）与删除（delete），保证操作原子性。
3. ‌**节点分裂与合并**‌
   * 当节点容量达到阈值时触发分裂（split），节点键过少时执行合并（merge）。
4. ‌**树的插入与删除**‌
   * 基于节点操作实现完整的树级数据增删逻辑。

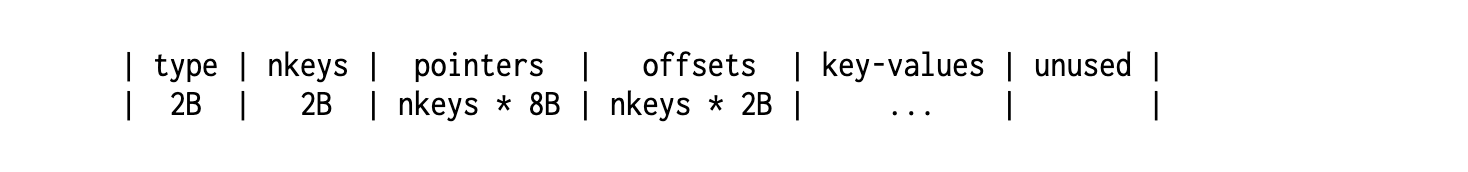
**B+树节点格式**

所有B+树节点的大小统一，以便后续使用空闲列表（free list）管理。尽管当前不涉及磁盘数据，但仍需明确具体的节点格式，因为它决定了以下两点：

1. 节点占用的字节大小。
2. 触发节点分裂的条件。

每个节点包含以下内容：

1. ‌**固定大小的头部**‌
   * 节点类型（叶节点或内部节点）。
   * 键的数量。
2. ‌**子节点指针列表**‌（仅内部节点）
3. ‌**键值对（KV pairs）列表**‌
4. ‌**键值对的偏移量列表**‌
   * 通过偏移量可对键值对执行二分查找



这是每个KV对的格式。长度后跟数据。

A close-up of a white background

AI-generated content may be incorrect.

我们的目标是学习基础知识，而非构建真实的数据库，因此采用了以下简化设计：

1. ‌**统一节点格式导致空间浪费**‌：叶节点（无需指针）与内部节点（无需存储值）使用相同的结构格式，造成冗余空间占用。
2. ‌**键数量简化**‌：内部节点包含 𝑛 个分支时存储 𝑛 个键，每个键复制自对应子树的最小键。而标准 B 树中，𝑛 分支仅需 𝑛−1 个键（通过相邻键分割子树）。此设计额外保留一个键以简化可视化理解。
3. ‌**节点容量限制**‌：
   * 固定节点大小为 4K（典型操作系统页大小），但键值（KV）可能任意增大而超出单节点容量。
   * 理论上可通过外部存储大型 KV 或动态调整节点大小解决，但这些实现细节非核心问题，故直接限制 KV 大小以保证其始终适配节点

**键的大小限制还确保内部节点始终可容纳至少两个键。**‌

**内存中的数据结构**

在代码实现中，‌**节点本质上是一段按格式解析的字节块**‌。这种设计使得数据从内存迁移到磁盘时无需序列化转换步骤，操作更简单直接。

**数据结构与 IO 解耦**

‌**内存数据结构和磁盘数据结构均需处理存储空间的分配与释放**‌。可通过回调函数抽象这一过程，形成数据结构与数据库其他组件之间的边界。

磁盘 B+树实现

数据库文件本质是一个由‌页号（即指针）‌引用的‌页面数组（节点数组）‌。我们通过以下回调函数实现页管理：

get: 从磁盘读取指定页

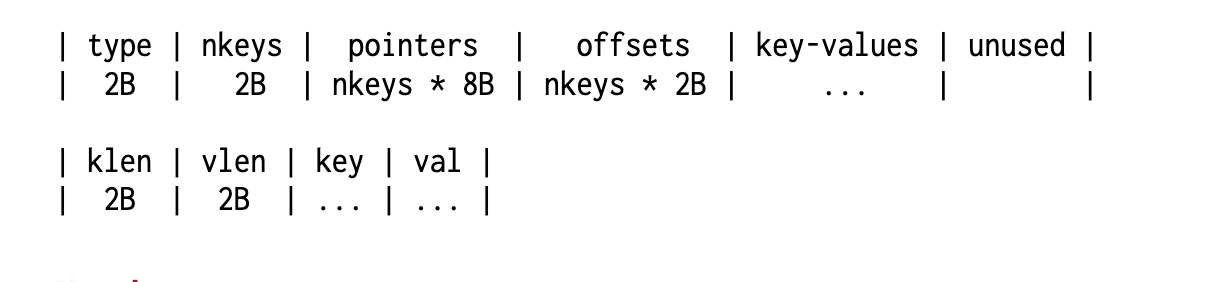
new: 分配新页并写入磁盘（‌写时复制‌机制）

del: 释放指定页

通过‌模拟回调函数‌，可对 B+树数据结构进行‌独立内存测试‌，无需依赖数据库其他组件。

4.2 解析节点存储格式

由于‌B+树节点本质是一段二进制数据块‌，我们将定义‌辅助函数‌（helper functions）来解析其内部结构并访问字段内容。



KV偏移量与键值对存储结构

数据格式采用‌连续紧密排列‌的方式存储所有键值对。虽然可通过逐个遍历键值对（KV）来定位第 n 个条目，但为了高效访问，我们在节点中设计了‌偏移量列表‌（offset list），实现 ‌𝑂(1)‌ 时间复杂度定位任意第 n 个 KV 对，并支持在节点内进行‌二分查找‌。

每个偏移量表示‌当前 KV 对的结束位置‌（相对于首个 KV 的起始位置）。由于首个 KV 的起始偏移量默认为 0，因此实际存储的是‌当前 KV 的结束偏移量‌，其数值也等于‌下一个 KV 的起始偏移量‌。

kvPos 返回第 n 个键值对（KV）相对于整个节点起始位置的‌偏移量‌。

该方法还通过‌**一次差一查询**‌（off-by-one lookup），便捷地返回‌**节点大小**‌（即已使用的空间）。

**节点内的 KV 查找机制**

‌**“seek”操作**‌（定位操作）同时服务于‌**范围查询**‌（range query）和‌**点查询**‌（point query），因此二者的底层逻辑本质上是相同的。

当前实现的函数名为 nodeLookupLE，因其使用‌**小于或等于运算符**‌（≤）。若需支持‌**点查询**‌，应改用‌**等于运算符**‌（=），此功能可作为后续优化步骤添加。

**4.2 更新 B+ 树节点**

**向叶子节点插入键**

‌**插入流程**‌：

1. ‌**定位插入位置**‌：通过 nodeLookupLE（基于 ≤ 运算符的查找）确定键的插入位置。
2. ‌**创建新节点**‌：将原节点的所有键值对（KVs）‌**按顺序拷贝到新节点**‌，并插入新增的键。此过程采用‌**写时复制**‌（Copy-on-Write）机制，避免直接修改原节点。

**节点拷贝函数**

* ‌**nodeAppendRange**‌：拷贝一段连续的键值对。
* ‌**nodeAppendKV**‌：拷贝单个键值对。  
  ‌**关键约束**‌：必须‌**按顺序调用**‌这两个函数，因为它们依赖前一个键值对的偏移量（Offset）来定位当前位置。

**更新内部节点**

1. ‌**子节点链接更新机制**‌：
   * 内部节点对子节点的引用（指针）始终通过‌**写时复制**‌（Copy-on-Write）方式更新。
   * 若子节点发生‌**分裂**‌（Split），父节点可能需要更新‌**多个子节点链接**‌（例如原链接替换为两个新子节点链接）。
2. ‌**节点分配接口**‌：
   * 子节点的分配需通过 tree.new 回调函数完成，该接口负责‌**创建新节点并返回其唯一标识**‌（如物理地址或内存指针）。

**4.3 B+ 树节点分裂**

**分裂规则**

* ‌**最小容量约束**‌：每个节点至少容纳 ‌**1 个键值对**‌（KV pair）。
* ‌**最坏情况处理**‌：
  + 当节点因插入超大键值对（例如单个 KV 超出节点容量）而无法直接二分分裂时，需进行 ‌**2 次分裂**‌。
  + 最终分裂为 ‌**3 个子节点**‌：原节点中间的超大键值对独立成新节点，剩余部分分裂为左右两个节点。

**临时节点状态**

* 分裂过程中生成的新节点‌**仅暂存于内存**‌（通过 nodeReplaceKidN 函数前的临时对象）。
* ‌**持久化时机**‌：实际存储分配需由 nodeReplaceKidN 函数完成（例如写入磁盘或内存池）。

**4.4 B+ 树插入操作**

**已实现的节点操作**

1. ‌**leafInsert**‌：向叶子节点插入新键值对（若键已存在则报错）。
2. ‌**nodeReplaceKidN**‌：更新内部节点的子节点链接（例如替换为分裂后的新子节点）。
3. ‌**nodeSplit3**‌：对容量超限的节点进行分裂（生成 3 个子节点）。

**完整插入流程**

1. ‌**从根节点逐层查找**‌：
   * 通过键值比较（key lookups）递归向下查找，直到定位到目标叶子节点。
2. ‌**更新叶子节点**‌：
   * 使用 leafInsert 插入新键值对。若键已存在，则调用 leafUpdate ‌**直接更新值**‌（而非插入重复键）。
3. ‌**递归更新父节点**‌：
   * 从叶子节点逐层向上回溯，对每个父节点调用 nodeReplaceKidN ‌**更新子节点链接**‌。
   * 若父节点因更新导致容量超限，则触发 nodeSplit3 分裂，并继续向上更新祖父节点。

**内存管理**

* ‌**临时节点生成**‌：每次操作生成的新节点（如分裂后的节点）需通过 tree.new 回调进行持久化分配。
* ‌**旧节点回收**‌：被替换的旧节点由 tree.dealloc 回调统一回收（例如释放内存或标记为可复用）。

**4.5 接下来的工作？**

工作已接近完成，我们只需在下一章添加以下内容：

1. ‌**节点合并与树删除**‌：实现节点合并逻辑，支持删除操作。
2. ‌**高级接口**‌：封装用户友好的 API（如 insert、delete 方法）。
3. ‌**用于测试的模拟节点回调**‌：提供测试用的伪回调函数，隔离存储层依赖。

**B+树删除操作与测试**

5.1 高级接口

我们将为 B+ 树添加键值存储（KV）接口。

大部分细节已在插入操作中介绍过，删除操作原理相似，因此不再赘述。若已理解原理，可跳过本章。

维护根节点的额外逻辑

在插入操作中维护根节点需注意：

根节点创建‌：若树为空，需创建初始根节点。

根节点分裂‌：若根节点分裂，需新增父根节点（树高度增加）。

哨兵值（Sentinel Value）

创建第一个根节点时有个技巧：插入一个空键（empty key）。其作用如下：

消除边界条件‌：空键是排序中的最小值，确保查找函数 nodeLookupLE 始终能找到有效位置。

修复查找逻辑‌：原 nodeLookupLE 在键超出节点范围时会失效，哨兵值强制所有实际键均大于空键，从而避免此问题。

关键细节

空键的作用‌：作为占位符，简化根节点初始化逻辑（无需特殊处理空树）。

根节点动态性‌：树的高度可能因插入/删除动态变化（例如根节点分裂或合并）。

**5.2 节点合并**

**节点更新函数**

为支持删除操作，需新增以下函数：

* ‌**shouldMerge**‌：判断是否需合并节点，并返回应合并的兄弟节点（左或右）。

**合并触发条件**

删除操作可能导致节点内容过少，需触发合并逻辑：

* ‌**空节点处理**‌：若节点为空且存在兄弟节点，则与兄弟节点合并。
* ‌**软性合并阈值**‌：为避免树中残留大量低利用率节点（如大部分空间未使用），采用‌**提前触发合并策略**‌——当节点容量低于页面大小的‌**1/4**‌时即触发合并（而非等到节点完全空），以此作为节点最小容量的软性限制。

**关键逻辑**

* ‌**合并方向决策**‌：shouldMerge 需动态选择合并的兄弟节点（左或右），优先选择能减少树结构调整的节点。
* ‌**空间利用率优化**‌：通过软阈值减少低效节点，避免树结构臃肿

**5.3 B+树删除**

**核心逻辑**

删除操作与插入操作类似，但核心逻辑从‌**分裂（splitting）**‌ 转为 ‌**合并（merging）**‌。

**合并条件与空节点处理**

* ‌**空节点合并限制**‌：即使某节点变为空节点，若其‌**无兄弟节点**‌（如根节点或唯一子节点），则‌**暂不合并**‌。
* ‌**空节点传递机制**‌：空节点会‌**上传至父节点**‌，等待后续合并机会（例如父节点触发合并条件时）。

**关键差异**

* ‌**动态调整方向**‌：插入操作可能导致树‌**高度增长**‌（根节点分裂），删除操作则可能导致树‌**高度收缩**‌（根节点合并）。
* ‌**延迟合并策略**‌：合并操作可能延迟至父节点处理，避免频繁结构调整

**5.4 测试 B+树**

**交互机制**

B+树数据结构仅通过 ‌**3 个页面管理回调函数**‌ 与数据库其他模块交互。

**内存模拟测试方法**

* ‌**页面模拟**‌：测试时可将页面（Page）‌**完全在内存中模拟**‌，无需真实磁盘操作。
* ‌**页面映射表**‌：
  + C.pages 是一个内存中维护的‌**已分配页面的映射表**‌（字典结构）。
  + 用于‌**验证指针合法性**‌及读取页面内容。
  + 指针本质上是‌**内存指针**‌，但 B+树代码无需感知其物理存储形式。

**测试流程与验证**

1. ‌**多场景更新**‌：对 B+树进行插入、删除等操作，覆盖各类边界条件。
2. ‌**通用验证**‌：
   * ‌**结构有效性验证**‌：
     + ‌**键有序性**‌：所有节点的键必须严格按升序排列。
     + ‌**节点容量限制**‌：节点实际存储的键值数量需满足 B+树的阶数约束（如最小/最大键数限制）。
   * ‌**数据一致性验证**‌：
     + 使用一个内存中的‌**参考映射（如 map 结构）**‌ 记录所有更新操作，验证 B+树内容与该映射完全一致。

**后续步骤**

* ‌**测试用例设计**‌：具体测试场景需自行设计（如并发操作、极端数据分布等）。
* ‌**磁盘存储延伸**‌：下一步将实现 B+树的‌**持久化存储**‌（基于磁盘的页面管理）

**仅追加键值存储**‌

**核心概念**

6.1 本章目标

我们将实现一个基于‌写时复制（Copy-on-Write）B+树‌的键值存储（KV Store），其数据持久化依赖于文件系统。

核心范围：‌持久性（Durability）‌ + ‌原子性（Atomicity）‌

仅追加文件（Append-Only）‌：文件内容仅追加写入，‌空间复用机制留到下一章实现‌。

单进程顺序访问‌：暂不处理并发问题，假设所有操作在单个进程中按顺序执行。

实现内容

本章将实现 B+树与磁盘页面交互的 ‌3 个核心回调函数‌：

（具体函数功能需根据上下文补充，例如：页面分配、读取、释放的回调逻辑）

6.2 两阶段更新：原子性与持久性

写时复制（Copy-on-Write）树的原子性

如第 3 章所述，‌写时复制 B+树通过原子更新根指针（Root Pointer）实现原子性‌，再通过 fsync 确保持久性。

但仅根指针的原子性不足以保证整个树的原子性。需满足以下条件：

新节点持久化先于根指针更新‌：所有新建或修改的节点必须先持久化到磁盘，才能更新根指针。

写入顺序 ≠ 持久化顺序‌：由于操作系统缓存（Page Cache）的存在，物理写入顺序无法保证。因此需通过 ‌两次 fsync‌ 强制持久化顺序：

第一次 fsync：确保所有新节点已持久化。

更新根指针。

第二次 fsync：确保根指针的更新持久化。

替代方案：基于日志（Log）的双写（Double-Write）

双写方案同样需要 ‌两阶段 fsync‌，但顺序不同：

写入更新页（带校验和）‌：将修改后的页面写入磁盘（临时区域）。

fsync 确保持久性‌：保证这些页面已落盘（用于崩溃恢复）。

原地更新数据‌：将双写区域的数据覆盖到原位置。

fsync 确保顺序性‌：确保步骤 3 的更新在步骤 1 之后持久化（后续可复用或删除双写数据）。

关键差异

双写 vs. 日志‌：

双写类似于日志（Log），每次更新仅需 1 次 fsync。

日志可批量缓冲多个更新（类似 LSM-Tree 的 WAL），提升性能。

写时复制的优势‌：

写时复制无需日志，因其通过‌版本化根指针‌天然支持原子性。

但日志仍具备通用优势（如批量缓冲、崩溃恢复），这也是日志在数据库中无处不在的原因之一。

内存数据的并发性

对于内存数据的‌原子性（Atomicity）‌（涉及并发访问），可通过‌互斥锁（Mutex）‌或‌原子 CPU 指令‌实现。但需注意一个类似问题：

内存读写顺序的不可预测性‌：由于 CPU 乱序执行（Out-of-Order Execution）等因素，内存读写操作的实际顺序可能与代码顺序不一致。

写时复制（Copy-on-Write）树的内存屏障

在内存中实现写时复制树时，需确保：

新节点对并发读线程可见性‌：在更新根指针（Root Pointer）之前，所有新节点必须对并发读线程可见。

内存屏障（Memory Barrier）的作用‌：

强制内存操作的顺序性，类似 fsync 对磁盘写入顺序的保障（但 fsync 还涉及持久化，而内存屏障仅关注可见性）。

同步原语的选择

互斥锁（Mutex）与系统调用‌：

使用互斥锁或任何操作系统提供的同步原语（如信号量）可‌以可移植的方式强制内存顺序‌。

避免直接依赖特定 CPU 的原子指令或内存屏障（因其对并发场景的支持有限且难以跨平台）。

关键结论

优先使用高级同步机制‌：操作系统提供的同步工具（如互斥锁）已隐式封装了底层内存屏障逻辑，兼顾可移植性与正确性。

避免过度依赖硬件特性‌：直接操作 CPU 原子指令或内存屏障需处理复杂的平台差异，且无法完全解决并发竞争问题。

**6.3 基于文件的数据库布局**

我们的数据库由单个文件构成，文件内容被划分为多个固定大小的‌**页（Page）**‌。

* ‌**页的类型**‌：
  + ‌**B+树节点页**‌：除第一页外，每个页对应一个 B+树节点。
  + ‌**元数据页（Meta Page）**‌：文件的第一页存储数据库的元信息，包括：
    - 指向当前最新根节点（Root Node）的指针。
    - 其他辅助数据（如版本号、校验信息等）。

这种设计通过‌**页编号寻址**‌实现高效读写，同时元数据页集中管理关键状态，便于崩溃恢复与一致性维护。

A close-up of a line

AI-generated content may be incorrect.

新增节点以日志形式追加与页数管理

新节点采用类似日志的‌追加写入‌方式，但存在一个问题：

文件大小不可靠性‌：系统掉电后，文件元数据（如文件大小）可能与实际数据内容不一致（例如文件系统可能未正确更新大小信息）。

解决方案‌：

元数据页显式记录页数‌：在元数据页（Meta Page）中维护当前总页数字段，替代依赖文件系统返回的文件大小。

原因‌：文件元数据与实际数据的同步行为与具体文件系统实现相关，显式记录页数可避免跨平台不一致问题。

目录的 fsync 操作

如第 1 章所述，‌文件重命名（Rename）后需对其父目录执行 fsync‌。此规则同样适用于‌新建文件‌场景，因为涉及两类持久化操作：

文件数据‌：新文件内容的写入。

目录条目‌：父目录中新增对该文件的引用。

具体实现策略

预防性目录 fsync‌：

当使用 O\_CREAT 标志‌潜在创建新文件‌时（即使文件已存在），在操作后主动对父目录执行 fsync。

目录 fsync 方法‌：以 O\_RDONLY（只读模式）打开目录文件描述符，对其调用 fsync。

关键意义

保障崩溃一致性‌：确保目录条目与文件数据的持久化顺序，防止掉电后出现"幽灵文件"（目录中记录存在但实际数据未落盘）或数据丢失。

文件系统特性处理‌：不同文件系统对元数据同步的保证不同，显式 fsync 可屏蔽底层差异。

**目录文件描述符与 openat 的竞态防护**

可通过 openat 使用目录文件描述符（fd）来打开目标文件：

* ‌**核心机制**‌：
  + 基于已打开的目录 fd（而非路径字符串）定位目标文件，‌**避免路径替换导致的竞态条件**‌。
  + 例如：若通过路径 /data/db 打开目录并获取 fd，后续即使 /data/db 被重命名或删除，openat 仍能通过原 fd 访问该目录下的文件

mmap、页面缓存与 IO 机制解析

mmap 允许将文件直接映射为进程的‌内存缓冲区‌，实现‌隐式磁盘 IO‌（访问即自动触发读写）。

操作系统分页基础

物理页‌：虚拟地址与物理地址映射的最小单元。

虚拟内存特性‌：进程的虚拟地址空间并非始终完全驻留物理内存，部分可被换出至磁盘。

当进程访问被换出的虚拟页时：

触发页错误（Page Fault）‌：CPU 中断进程，将控制权移交操作系统。

操作系统处理流程‌：

a. 从磁盘加载被换出的数据至物理内存。

b. 重映射虚拟地址至新加载的物理页。

c. 恢复进程执行。

进程恢复‌：虚拟地址已指向真实的物理内存，访问继续。

mmap 的运作原理

映射地址范围‌：mmap 为进程分配一段虚拟地址范围，初始无物理内存或磁盘数据支撑。

读场景自动 IO‌：

进程首次访问映射范围内的某页 → 触发页错误。

操作系统将对应文件数据读入‌页面缓存（Page Cache）‌，并重映射该页至缓存。

后续访问直接命中缓存，无需磁盘 IO。

写场景脏页标记‌：

CPU 监测到进程修改映射页 → 标记该页为‌脏页（Dirty Bit）‌。

操作系统后续将脏页异步写回磁盘，fsync 可强制同步并等待写完成。

mmap 与传统写操作的对比

Linux 写操作（write）‌：数据先写入页面缓存，由系统异步刷盘。

mmap 写操作‌：直接修改映射页（即页面缓存），脏页标记与回写机制与 write 共用同一缓存层。

本质一致性‌：两者最终均通过页面缓存与磁盘交互，性能差异主要源于系统调用开销。

技术选型建议

非强制使用‌：开发者可选择 mmap 或传统读写接口。

核心价值‌：理解底层机制有助于优化 IO 密集型应用的性能与一致性策略。

6.4 磁盘页管理实现

我们将利用 ‌mmap‌ 实现磁盘页管理回调函数，主要因其‌便捷性‌。

mmap 的调用方式

文件支持的映射类型‌：

只读映射‌：使用 PROT\_READ（内存保护标志，允许读访问）与 MAP\_SHARED（映射共享模式）标志。

读写映射‌：需结合 PROT\_READ | PROT\_WRITE 与 MAP\_SHARED。

写时复制（Copy-on-Write）‌：使用 MAP\_PRIVATE 标志（进程私有映射）。

映射范围与文件扩展‌：

映射范围可大于当前文件大小‌：即使文件初始较小，mmap 允许映射一个更大的虚拟地址空间。

利用动态扩展‌：当文件后续增长时，操作系统会自动将新增部分纳入已映射的地址范围，无需重新映射。

核心优势

简化扩容逻辑‌：文件动态增长时，映射区域自动覆盖新数据，避免手动调整映射或处理碎片。

零拷贝高效访问‌：通过直接操作内存地址访问文件内容，省去用户态与内核态间的数据复制开销。

mmap 处理动态增长文件的策略

mremap 的局限性‌：

功能类比‌：mremap 类似于内存的 realloc，可扩展映射的虚拟地址范围。

地址变更问题‌：重新映射可能导致虚拟地址改变，影响后续并发读取的稳定性（后续章节会涉及）。

解决方案：增量映射‌

动态添加新映射‌：每当文件扩展时，新增一个映射覆盖扩展部分。

性能隐患‌：频繁新增映射会导致操作系统维护大量映射条目，增加管理开销。

优化手段：指数级扩展‌

减少映射次数‌：通过指数级扩大每次映射的增量（例如每次翻倍），显著降低总映射次数。

利用 mmap 特性‌：映射范围可预先设定远超当前文件大小，操作系统仅在实际访问时分配物理资源。

极端场景：预映射超大地址空间‌

虚拟地址无实际成本‌：在 64 位系统中，即使声明一个极大映射（如 1TB），未实际使用的虚拟地址不会占用物理内存或磁盘资源。

适用性‌：此方法适用于‌小型数据库或实验性系统‌，简化动态扩容逻辑，但需注意实际文件增长与地址管理需求。

设计权衡‌

灵活性 vs 性能‌：

增量映射‌：灵活适应文件动态增长，但需谨慎控制映射频率。

预映射超大空间‌：规避扩容问题，但需确保系统地址空间充足（64 位系统无压力）。

实践建议‌：根据文件增长模式选择策略，优先利用 mmap 的自动按需加载特性。

页面更新捕获机制

B+树更新与页面分配‌：

BTree.new 回调函数‌：在 B+树更新过程中收集新生成的页面。

页号分配策略‌：从数据库文件（DB）的末尾动态分配新页号，确保追加写入时不覆盖已有数据。

持久化操作‌：所有新页面会在 B+树更新完成后，以‌追加（append）‌方式写入数据库文件。

pwritev 系统调用的作用‌

功能特性‌：

偏移量控制‌：允许指定写入文件的偏移位置（需手动管理，因为后续还需更新元数据页）。

多缓冲区聚合写入‌：通过多个非连续输入缓冲区（input buffers）一次性提交数据，由内核（kernel）合并后写入磁盘，减少系统调用次数。

使用场景‌：适用于需要精确控制写入位置且数据分散在多个内存区域的场景（如同时更新数据页和元信息页）。

设计关键点‌

写入顺序控制‌：需确保数据页先写入文件，元数据页最后更新，避免因崩溃导致元数据与数据不一致。

性能优化‌：利用 pwritev 的聚合写入特性，降低频繁小数据块写入的开销。

6.5 元页面（Meta Page）

元页面的读取‌：

数据库启动时首先读取元页面，用于获取文件的基础信息（如版本、根节点位置等）。

文件类型标识‌：

在元页面中设置‌魔数字节（Magic Bytes）‌，通过特定字节序列唯一标识文件类型（例如自定义数据库格式）。

text

Copy Code

┌───────────────┬───────────────┐

│ Magic Bytes │ 其他元数据 │

│ (固定值，如 │ (如根节点页号) │

│ 0x1234ABCD) │ │

└───────────────┴───────────────┘

空文件处理‌：

保留元页面‌：若文件为空（如新创建的数据库文件），则预分配元页面的空间（通常为固定大小，例如4KB），避免后续写入时频繁调整文件大小。

初始化操作‌：在保留的元页面中写入初始魔数字节和默认元数据，标记文件为有效数据库文件。

关键作用‌

一致性校验‌：通过魔数字节快速识别文件是否损坏或格式错误。

快速恢复‌：元页面存储关键指针（如根节点页号），崩溃后可直接定位数据起点。

元页面（Meta Page）更新机制

硬件级原子性（Power-Loss Atomicity）‌

小数据块原子写入‌：

向磁盘写入‌页对齐的小数据‌（如单个扇区大小）时，硬件层面可能在‌掉电时保证原子性‌（要么全写，要么不写）。部分真实数据库依赖此特性实现元页面的安全更新。

实现方式‌：直接修改元页面所在扇区，确保单次写入操作覆盖整个扇区。

原子性的多层次差异‌

系统调用层非原子性‌：

并发读写冲突‌：write 系统调用在‌并发读写下不保证原子性‌（例如读者可能读到部分更新的数据）。

页缓存影响‌：文件系统的页缓存机制可能导致读写可见性延迟，进一步破坏原子性。

解决方案‌：

LSM 树（Log-Structured Merge Tree）启发‌：

内存化并发控制‌：将更新操作限制在内存中的 MemTable（如 LSM 树的第一层），避免直接并发读写磁盘。

元页面内存副本‌：在内存中维护元页面的副本，通过‌互斥锁（Mutex）‌同步访问，确保并发安全。

无硬件原子性时的保障措施‌

日志 + 校验和（Log + Checksum）‌：

双缓冲（Double Buffering）‌：

交替写入‌：维护两个带校验和的元页面（如 meta\_page\_A 和 meta\_page\_B），每次更新时交替写入，确保至少一个元页面在掉电后有效。

旋转日志（Rotating Log）‌：类似仅保留两个日志项的环形缓冲区，循环覆盖旧数据。

text

Copy Code

┌───────────────┐ ┌───────────────┐

│ meta\_page\_A │ ←→ │ meta\_page\_B │

│ (checksum\_1) │ │ (checksum\_2) │

└───────────────┘ └───────────────┘

操作流程‌：

步骤 1‌：更新 meta\_page\_A 并写入磁盘，记录校验和。

步骤 2‌：下次更新时，改为写入 meta\_page\_B，覆盖前一次操作的 meta\_page\_A。

恢复逻辑‌：重启后检查两个元页面的校验和，选择有效版本。

关键设计权衡‌

性能与可靠性‌：双缓冲增加了写入次数，但通过校验和和交替写入提高了数据可靠性。

内存开销‌：内存中维护元页面副本需占用额外空间，但避免了磁盘并发控制的复杂性。

硬件依赖性‌：优先利用硬件原子性（如支持单扇区原子写入的磁盘），否则回退到日志+校验和方案。

总结‌

硬件原子性优先‌：在支持单扇区原子写入的硬件上，直接更新元页面。

无硬件支持时‌：通过双缓冲 + 校验和模拟原子性，确保崩溃后至少存在一个有效元页面。

内存化并发控制‌：将元页面同步问题转移到内存，通过互斥锁和副本机制规避磁盘并发读写风险。

6.6 错误处理（Error Handling）

I/O 错误后的场景与应对策略‌

基础错误传播‌

最低要求‌：通过 if err != nil 向上传播错误（例如调用方需显式检查错误）。

I/O 错误后的数据库使用可能性‌

更新失败后的读取行为‌：

合理选择‌：数据库应表现得像‌未发生任何更改‌（例如读取操作返回更新前的数据）。

原理‌：若写操作未确认成功（如未调用 fsync 或写入失败），则数据未持久化，需保证读操作的原子视图。

失败后重试更新‌：

错误持续‌：若底层错误未修复（如磁盘损坏），再次更新‌预期仍会失败‌。

临时错误‌：若错误是暂时的（如网络闪断、磁盘缓冲区满），可尝试‌自动重试或等待恢复‌。

设计建议‌：加入重试次数限制和退避策略（如指数退避）。

问题解决后重启数据库‌：

本质‌：此场景属于‌崩溃恢复（Crash Recovery）‌，已在第 3 章讨论。

恢复逻辑‌：

检查元页面和日志的校验和。

回滚未提交的事务，重放已提交但未刷盘的日志。

关键设计原则‌

状态一致性‌：

确保任何 I/O 错误后，数据库对外呈现‌一致的逻辑状态‌（例如未提交的写操作对读者不可见）。

错误隔离性‌：

单次 I/O 错误不应导致数据库进入不可恢复状态（如元页面损坏）。

通过‌预写日志（WAL）‌或‌双缓冲元页面‌隔离错误影响范围。

明确错误层级‌：

可恢复错误‌（如临时磁盘空间不足）：记录日志并重试。

不可恢复错误‌（如硬件损坏）：向上抛出致命错误，终止服务。

错误处理流程示例‌

text

Copy Code

写请求 → 尝试写入磁盘

├── 成功 → 返回成功

└── 失败 → 判断错误类型

├── 临时错误 → 等待并重试（最多 N 次）

└── 永久错误 → 标记数据库为不可用，触发告警

总结‌

静默与显式‌：读取失败后应静默返回旧数据，写入失败需显式通知调用方。

状态无损‌：通过日志和校验机制确保即使崩溃也能恢复到一致状态。

分级处理‌：区分临时与永久错误，避免无意义的重试或掩盖严重问题。

**回滚到旧版本**

存在关于 fsync 失败处理的调研[3]，结论是处理方式‌**依赖具体文件系统**‌。若在 fsync 失败后尝试读取数据，部分文件系统会返回失败的数据（因页缓存与磁盘内容不一致），导致读取失败写入的数据不可靠。

但由于我们采用‌**写时复制（Copy-on-Write）**‌，此问题可被规避：通过回滚到旧的树根指针（tree root）即可跳过问题数据。虽然树根指针存储在元页面中，但数据库打开后‌**不会从磁盘重新读取元页面**‌，因此只需回滚内存中的根指针即可实现安全恢复。

写入失败后的处理

只读模式支持‌：写入失败后，数据库仍可进入‌只读模式‌继续提供读取服务。

读取失败风险‌：若使用 mmap，发生读取错误时，进程会被 SIGBUS 信号直接终止（这是 mmap 的缺点之一）。

临时写入错误的恢复

错误类型与影响‌：

临时错误‌（如“磁盘空间不足”）：若首次更新失败但后续更新成功，最终状态仍有效。

中间状态问题‌：两次更新之间，磁盘上的元页面（meta page）内容‌状态未知‌，可能导致崩溃后数据损坏。

崩溃恢复逻辑‌：

元页面与内存不一致‌：若 fsync 在元页面更新时失败，磁盘上的元页面可能是‌新版本或旧版本‌，而内存中的树根指针仍为旧版本。

覆盖风险‌：后续成功更新可能覆盖新版本的数据页，导致崩溃后残留‌损坏的中间状态‌。

解决方案‌：在恢复时‌重写最后一个已知有效的元页面‌，确保磁盘状态与内存一致。

错误处理的可靠性问题

我们依赖文件系统正确报告错误，但有证据[4]表明其不可靠。因此，‌整个系统能否妥善处理错误仍存疑‌。

6.7 仅追加（Append-only）KV存储总结

文件布局‌：基于写时复制的 B+ 树设计。

持久性与原子性‌：通过 fsync 保证。

错误处理‌：如前述策略。

关键改进‌：磁盘上的 B+ 树是核心进展，仅需补充‌空闲列表（Free List）‌即可实用化。

**空闲列表：回收与重用**‌

第七章 空闲列表：回收与重用‌

实现 KV 存储的最后一步是‌复用已删除的页面‌，这也是内存数据结构的常见问题。

7.1 内存管理技术‌

目标‌：

内存（空间）管理可分为‌手动‌或‌自动‌。垃圾回收器（GC）是自动的，无需程序员干预即可检测未使用对象。

核心问题在于如何‌处理（复用）未使用对象‌。

为何无需 GC‌：

在树形数据结构中，检测未使用节点是简单的（例如通过 BTree.del 回调函数实现）。

需重新实现这些回调逻辑以管理空闲资源。

空闲对象列表‌：

磁盘空间按‌固定大小页面‌管理的原因：删除后的页面可互换复用（类似数据库按需分配页）。

相较于 malloc 等处理任意大小的内存管理，固定页管理更简单。

空闲列表（Free List）的实现‌：

需维护一个‌未使用页面的列表‌（称为“空闲列表”或“对象池”）。

对于内存数据，可直接用指针数组或嵌入对象的链表实现。

嵌入式链表‌

最简单的方案是使用‌嵌入式（侵入式）链表‌。链表指针直接嵌入对象内部，借用对象自身的空间，无需为数据结构分配额外内存。

但此方案与‌写时复制（Copy-on-Write）‌冲突，因更新时会覆盖数据。

外部列表‌

另一种方案是将未使用页面的指针存储在‌外部数据结构‌中。但外部数据结构本身占用空间，这一问题后续需解决。

例如，空闲列表可设计为记录未使用页号的日志，添加条目仅需追加。核心问题是如何删除条目以避免日志无限增长。

7.2 磁盘上的链表‌

空闲列表的需求‌

问题场景‌：

若将空闲列表视为‌顺序记录的日志‌，在‌写时复制（Copy-on-Write）树‌中，每次更新需创建新节点并删除旧节点，导致每次操作既向空闲列表‌添加‌条目，又从中‌删除‌条目。

若从列表‌尾部删除‌条目，新条目会覆盖旧数据，需依赖第3章讨论的额外崩溃恢复机制。

若从列表‌头部删除‌条目，如何回收已删除条目的空间？问题回归原点。

解决方案‌：

空闲列表需设计为‌基于页的链表‌（类似 B+树节点），每个页可存储多个条目，称为‌展开式链表（unrolled linked list）‌。

该结构实现‌自管理‌：

新增节点‌时，优先从自身空闲列表中获取页。

删除的链表节点‌会被重新加入空闲列表以供复用。

设计要点‌

页面特性‌：

每个页包含多个条目（页号）。

页内采用‌原地更新‌，但条目仍以‌仅追加（append-only）‌方式写入。

操作逻辑‌：

条目从‌尾部节点追加‌，从‌头部节点消费‌。

此设计便于保持尾部节点的‌仅追加特性‌，简化实现。

**空闲列表的磁盘布局**‌

* 每个节点以‌**指向下一节点的指针**‌开头，条目紧随其后追加。
* ‌**元页（meta page）**‌中存储头节点和尾节点的指针，其中尾节点指针用于支持 ‌**O(1) 时间复杂度**‌的插入操作

更新空闲列表节点‌

无空闲列表时‌：仅‌元页（meta page）‌采用原地更新，这是写时复制（Copy-on-Write）实现崩溃恢复简化的关键。

引入空闲列表后‌：新增两类原地更新操作：

链表节点的‌下一节点指针‌。

页内‌追加的条目‌。

崩溃恢复分析‌：

虽然链表节点原地更新，但页内数据‌不会被覆盖‌。若更新中断，元页仍指向原数据，无需额外崩溃恢复机制。

与元页不同，此类更新‌无需保证原子性‌。

嵌入式链表的可行性‌：

若在 B+树节点中‌预留下一节点指针‌，嵌入式链表仍可工作（尽管此设计会导致‌写入放大（write amplification）加倍‌）。

此方案可脱离原书设计，灵活调整。

7.3 空闲列表的实现‌

页面管理回调函数‌

与 B+树类似，通过以下回调隔离页面管理：

get‌：读取页面（与之前相同）。

new‌：新增页面（原用于 B+树）。

set‌：返回可写缓冲区以捕获原地更新。

del‌：无需此操作，因空闲列表‌自主管理‌释放的页面。

数据结构设计‌

每个节点包含‌可变数量的条目‌（上限为 FREE\_LIST\_CAP），需记录：

头序列号（headSeq）‌：头节点中第一个条目的位置索引。

尾序列号（tailSeq）‌：尾节点中条目结束的位置索引。

序列号特性‌：

headSeq 和 tailSeq 为‌单调递增‌的数值，而非循环索引。

通过取模运算（索引 = 序列号 % 节点容量）计算实际索引。

单调递增确保其成为链表位置的‌唯一标识符‌，并可通过比较二者防止‌头节点覆盖尾节点‌。

更新时序控制‌

初始状态‌：更新开始时，将当前 tailSeq 保存为 maxSeq（本次更新的最大可用序列号）。

更新限制‌：本次更新中，headSeq 的推进‌不可超过‌ maxSeq，防止覆盖本次新增的尾部条目。

状态同步‌：下次更新开始时，将 maxSeq 更新为新的 tailSeq，确保后续操作基于最新尾部位置。

循环机制‌：通过 maxSeq 的阶段性推进，实现跨更新操作的‌时序隔离‌。

空闲列表的消费机制‌

移除条目‌：

从头部节点移除条目时，仅需‌递增头序列号（headSeq）‌。

若头部节点条目耗尽（headSeq 超出节点容量），则‌切换到下一节点‌。

自我管理逻辑‌：

空闲列表具备‌自我管理能力‌：移除的头节点会重新加入自身，供后续分配使用。

空链表处理策略‌：

若最后一个节点被移除，链表将处于“空链表”状态，需处理特殊边界条件。

实践表明‌：设计链表始终保持‌至少一个节点‌，可规避边界条件（如通过 assert(fl.headPage != 0) 断言确保头节点非空）。

空闲列表的压入操作‌

条目追加‌：

向尾节点追加条目时，仅需‌递增尾序列号（tailSeq）‌。

若尾节点已满，则‌立即追加一个空尾节点‌，确保链表始终至少保留1个节点（防止原尾节点被移除后成为头节点导致空链表）。

自我管理逻辑‌：

空闲列表‌优先尝试从自身回收的节点中获取新尾节点‌，仅当无可用节点时才执行追加操作。

7.4 基于空闲列表的键值存储（KV）‌

页面管理机制‌

页面重用‌：

重用页面时，‌原地覆写‌（in-place），需通过‌待处理更新映射表‌（pending updates map）追踪变更。

页面分配‌：

BTree.new 替换为 KV.pageAlloc，分配逻辑为：‌优先从空闲列表获取页面‌，不足时再追加新页面。

页面写入‌：

KV.pageWrite 返回‌可写页面副本‌，用于捕获原地更新（避免直接修改原始数据）。

页面读取‌：

若需读取已更新的页面，KV.pageRead ‌优先查询待处理更新映射表‌，确保获取最新数据。

元数据页面更新‌

元数据结构‌：

元数据页面新增‌空闲列表指针（头/尾）‌，与树根指针（root\_ptr）‌原子更新‌，结构如下：

plaintext

Copy Code

| 字段 | 长度 |

|--------------|--------|

| sig (签名) | 16B |

| root\_ptr | 8B |

| page\_used | 8B |

| head\_page | 8B |

| head\_seq | 8B |

| tail\_page | 8B |

| tail\_seq | 8B |

空闲列表初始化‌：

空数据库初始化时‌，空闲列表默认包含至少1个节点，并分配一个空节点（empty node）。

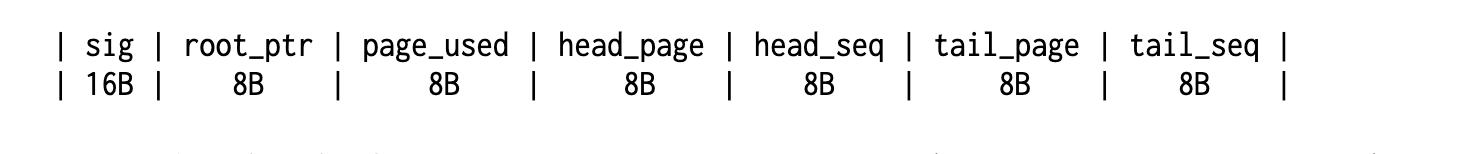
通过断言 assert(fl.headPage != 0) 确保头节点页号非空。

页面重用逻辑‌：

headSeq 受 maxSeq 限制，‌更新操作间将 maxSeq 设为 tailSeq‌，以允许重用上一版本的页面。

并发扩展‌：

本章假设‌顺序访问‌，后续支持并发时，headSeq 将改为受‌最旧未完成读操作‌的版本号限制。



7.5 KV存储的阶段性总结‌

已实现核心功能‌

存储结构‌：

设计了‌写时复制（Copy-on-Write）的B+树‌文件布局。

通过 fsync 机制保障‌持久性（Durability）与原子性（Atomicity）‌。

基于‌空闲列表（Free List）‌实现了磁盘页面的动态管理。

当前能力范围‌

这足以支撑基础的键值存储功能（get/set/del）。

第二部分扩展方向‌

进阶能力‌：

基于KV存储实现‌关系型数据库‌（Relational DB）。

支持‌并发事务处理‌（Concurrent Transactions）。

**基于KV存储的表结构**‌

8.1 将数据行编码为键值对‌

索引查询的核心操作‌

在关系型数据库中，数据以二维表（行与列）形式建模。用户通过SQL声明查询意图，数据库则“神奇地”返回结果。但需注意：

并非所有查询都适用于OLTP场景‌ —— OLTP要求通过合理的‌表结构设计‌与‌索引规划‌，确保查询的高效性与可扩展性。

索引查询的本质‌

点查询（Point Query）‌：

通过‌指定键‌快速定位单行数据。

范围查询（Range Query）‌：

通过‌键范围‌筛选数据，并按顺序遍历结果集。

数据结构选择‌

B+树‌与‌LSM树‌因天然支持‌有序范围遍历‌，成为主流索引结构。

哈希表‌因无法高效处理范围查询，通常被排除。

主键作为“键”‌

点查询场景‌

定位单行数据需‌唯一标识符‌，即由部分列构成的‌主键（Primary Key）‌。

键值映射规则‌

主键列‌直接作为键（Key），‌其他列‌归入值（Value）。

无主键表的处理‌

若表未显式定义主键，数据库会‌自动添加隐式自增主键‌（Hidden, Auto-generated Primary Key）。

二级索引的独立表逻辑‌

多维度索引机制‌

除主键外，表可通过‌二级索引‌支持多种查询方式。

二级索引通过‌间接引用‌实现：‌索引键‌映射到唯一行标识符（即主键）。

主键与索引的关系‌

主键本质为‌自带唯一约束的索引‌。

每个二级索引逻辑上类似‌独立表‌，其键为“索引列+主键”，值为空或关联数据。

替代方案：自动生成行ID‌

隐式主键设计‌

部分数据库将‌自动生成的ID‌作为“真实”主键（如自增整数），而非用户显式定义的主键。

此时，‌用户主键‌仅作为逻辑标识，主键与二级索引均需通过自动生成的ID间接映射。

核心优势‌

自动生成ID‌为短小、定长的整数，而用户主键可自由定义（如长字符串）。

带来的优化：

索引结构更紧凑‌：内部节点可存储更多键值（树高度更低）。

二级索引体积更小‌：无需冗余存储用户主键内容。

**8.2 数据库表结构设计‌**

**‌表前缀机制‌**

* 数据库中的‌**多表与索引**‌共享单一B+树：为所有键‌**预加自动生成的前缀**‌，避免维护多棵树的开销。
* ‌**前缀设计选项**‌：
  + ‌**32位自增整数**‌：定长、高效。
  + ‌**表名**‌：直接可读，但长度不可控（存储与性能开销）

**数据类型‌**

* ‌**关系型数据库相对KV的优势**‌：支持更丰富的数据类型。此处实现‌**两种基础类型**‌：字符串（String）与整数（Integer）。
* ‌**单元格值（Cell Value）**‌：基于类型的带标签联合结构（Tagged Union）。

**‌记录（Records）‌**

* 表示‌**列名与值的有序集合**‌。

**‌模式（Schemas）‌**

* 本章仅定义‌**主键**‌，索引设计后续讨论。

内部表

表的模式(schema)存储在哪里？‌

既然我们在编写一个数据库，知道如何存储数据，我们将其存储在一个预定义的‌内部表‌中。

def 列是 JSON 序列化的 TableDef（表的定义信息）。

其他元信息如何管理？‌

我们还需要维护一些额外信息，例如一个‌自增计数器‌（用于生成表前缀）。为此，我们需定义另一个内部表来存储这些元信息。

8.3 查询、更新、插入、删除、创建

点查询与更新接口‌

单行数据读写接口定义‌：

DB 是 KV 存储的封装‌：

通过主键（Primary Key）进行查询，rec 参数既是输入的主键值，也是输出的完整行数据。

列字段处理‌：

处理列字段的代码较为简单，此处略过。

数据编解码‌：

下一步是数据的序列化与反序列化，可采用任意编码方案（如 JSON、Protobuf 等）。

用户表模式（Schema）的读取逻辑

根据表名获取模式‌：

用户接口通过‌表名‌操作表，因此需先查询内部表（如 @table）获取该表的模式（Schema）。

模式以 ‌JSON 序列化的 TableDef‌ 格式存储。

模式缓存优化‌：

建议在内存中缓存表结构，减少对内部表的查询次数。

合理性‌：实际应用中表数量通常有限，缓存不会显著占用内存。

插入或更新行

三种 SQL 更新语句的区别‌（根据主键是否已存在）：

INSERT‌：仅添加新行（主键不存在时生效）。

UPDATE‌：仅修改已存在的行（主键必须存在）。

UPSERT‌：不存在则插入，存在则更新（PostgreSQL 特性；MySQL 为 ON DUPLICATED KEY UPDATE，SQLite 为 INSERT OR REPLACE）。

底层实现方式‌：

通过扩展 ‌BTree 插入方法‌，增加模式标志（mode flag）区分操作类型。

更新函数仅处理‌完整行数据‌，部分更新（如读-改-写）需在更高层（如查询语言）实现。

创建表的流程

查重‌：检查 @table 中是否存在同名表。

读取计数器‌：从 @meta 读取表前缀计数器的当前值。

更新计数器‌：递增并更新 @meta 中的计数器值。

插入表结构‌：将表结构写入 @table。

问题‌：

涉及更新两个键，因此失去原子性，后续通过事务机制修复。

4. 基于键值存储（KV）实现表的结论

在键值存储上实现表结构并无本质差异，核心区别在于需额外处理 ‌数据序列化‌ 和 ‌表结构维护‌。但当前工作尚未完成，后续需实现：

范围查询‌（Range Queries）

二级索引‌（Secondary Indexes）

范围查询

9.1 B+树迭代器

迭代器接口‌

核心操作为 ‌定位（seek）‌ 和 ‌迭代遍历（iterate）‌，用于范围查询。B+树的位置通过有状态的迭代器 BIter 表示。

示例‌：

查询 a <= key 的实现逻辑如下：

使用 seek 定位到第一个 >= a 的键。

通过 iterate 遍历后续键，直至超出范围。

遍历B+树

定位与回溯机制‌

当前键的位置‌：用于查找节点内的相邻键（兄弟键）。

跨节点查找‌：若兄弟键位于相邻节点，需沿 ‌根节点到叶节点的完整路径‌ 回溯至父节点。

迭代器移动逻辑‌：类似逐位进位（如数字递增），逐层调整路径。

说明‌：

由于未使用父节点指针，遍历需维护完整的节点路径栈。

定位键值（Seek）

机制说明‌

点查询逻辑‌：定位键值类似点查询（point query），需 ‌记录遍历路径‌。

运算符支持‌：nodeLookupLE 用于 ≤ 条件，还需实现其他比较运算符（如 ≥、= 等）。

9.2 保序编码

处理任意类型数据的字节序列化‌

核心需求‌：B+树支持任意字节的字符串键，但列值可为数字、多列组合键等类型。为支持范围查询，序列化后的键必须 ‌保留原数据类型的比较语义‌。

实现方案‌

回调函数比较‌：替换 bytes.Compare，根据表结构解码并比较键值（需额外计算开销）。

保序序列化‌：采用特殊编码格式，使序列化后的字节 ‌直接反映数据类型的排序顺序‌。

选择‌：采用方案 2（保序序列化）作为快捷实现路径。

数字编码

无符号整数‌

保序编码方案‌：使用 ‌大端序（Big-Endian）‌ 存储，高位字节在前，直接通过 bytes.Compare 比较即可反映数值顺序。

示例：

0x0000000000000001 → 00 00 00 00 00 00 00 01

0x0000000000000100 → 00 00 00 00 00 00 01 00

有符号整数‌

补码表示法‌：负数通过补码编码，需 ‌交换正负半区顺序‌ 以保序，即 ‌翻转最高位（MSB）‌。

原理‌：

正数最高位置 0，负数最高位置 1。通过翻转最高位，使负数编码值小于正数，确保排序正确。

核心思路

高位优先编码‌：按大端序排列数据（高位字节/位在前）。

位模式映射‌：将数据位模式转换为无符号整数，使其自然排序与数据逻辑顺序一致。

读者练习（浮点数保序编码）‌

转换思路‌：浮点数由 ‌符号位 + 尾数 + 指数‌ 组成，需调整三部分位模式使其：

符号位‌：负数编码值小于正数（如将符号位取反）。

指数/尾数‌：按高位优先重组（类似整数的大端序处理）。

字符串列编码

多列拼接问题‌

直接拼接字符串列会导致歧义（如 ("a", "bc") 与 ("ab", "c") 无法区分）。需通过 ‌编码分隔符‌ 明确列边界。

编码方案‌

定长前缀‌：前置长度信息（需解码）。

分隔符标记‌：用终止符（如空字节 0x00）分隔各列。

示例：

("a", "bc") → "a\x00bc\x00"

("ab", "c") → "ab\x00c\x00"

分隔符转义规则‌

问题‌：输入可能包含分隔符 0x00，需通过 ‌转义字节 0x01‌ 处理冲突：

0x00 → 0x01 0x01

0x01 → 0x01 0x02

特性‌：转义后的序列仍保持原字符的排序顺序。

元组编码

多列保序规则‌：元组按列逐项比较（类似字典序），每列需独立编码后 ‌无歧义拼接‌。

实现方式‌：将每列的保序字节序列直接拼接，确保：

各列编码自身保序（如大端整数、转义分隔的字符串）。

列间字节无重叠（如定长列或分隔符明确边界）。

范围查询实现

扫描器（Scanner）‌：封装 B+树迭代器，将键值对（KV）解码为数据行。

主键区间输入‌：指定主键的起止范围（闭区间或开区间）。

开区间处理‌：将 Key2 设为系统支持的最大值（上界开区间）或最小值（下界开区间）。

示例：查询 [start, +∞) → 实际范围设为 [start, MAX\_VALUE]。

核心要点

B+树迭代器‌：实现数据遍历机制。

保序编码‌：确保存储字节序与数据逻辑序一致。

下一步‌：通过 ‌额外表结构‌ 支持二级索引。

二级索引

核心要点

二级索引本质‌：存储为 ‌额外键值对‌，包含 ‌主键值‌（例如：索引键 → 主键）。

索引标识‌：每个索引通过 ‌键前缀‌（如 idx\_1:）在 B+树中唯一区分。

主键与索引关系‌：第一个索引自动作为 ‌主键‌，因其本身已是唯一索引。

核心要点

二级索引结构‌

将 ‌主键‌ 作为 B+树 ‌值‌（索引键 → 主键），用于回表查询完整数据。

但二级索引允许 ‌重复键‌（无唯一性约束），直接存储会导致键冲突。

解决方案‌

不修改 B+树‌，而是将 ‌主键追加到索引键‌（复合键 = 索引键 + 主键），使键唯一，值留空。

例如：索引键 (age, id)，通过 age 查询时，按复合键前缀匹配。

核心要点

索引选择规则‌：通过 ‌列匹配‌ 确定适用的二级索引，范围查询需索引与查询键匹配。

索引存储位置‌：选中的索引存储在 Scanner 类型中，供 Scanner.Deref() 调用。

多列索引应用‌：例如索引 (𝑎, 𝑏) 支持查询 (𝑎, 𝑏) > (1, 2)，也可支持 𝑎 > 1（等效于 (𝑎, 𝑏) > (1, +∞)）。

缺失列编码为无穷大‌

查询仅使用索引部分列时（如索引 (𝑎, 𝑏) 但查询 𝑎 > 1），需将未使用的列编码为 ‌+∞‌（\xff）或 ‌-∞‌（空字符串 ""）。

-∞ 处理‌：若列缺失且需表示下限（如 𝑏 > -∞），可直接忽略（因数据列编码不为空）。

+∞ 处理‌：为每个编码列 ‌前置标签‌（如 0x01），避免与 \xff 冲突，确保 \xff 仅用于显式无穷大标记。

核心要点

列类型标签‌：在编码时为每列 ‌前置类型标识码‌（如 0x01 表示整型），便于调试时通过 hexdump 直接解码数据。

前缀查询支持‌：通过添加类型标签，确保前缀列（如索引 (a, b) 中的 a）的范围查询逻辑正确性。

多键更新的原子性

原子性不可组合！‌ 当涉及多个键时，即使单个 KV 操作是原子的，我们也会失去原子性。

崩溃回滚要求‌：如果在更新二级索引时数据库崩溃或发生错误，它应回退到之前的状态。

简单 KV 接口的局限性‌：仅通过 get、set、del 实现原子多键操作非常困难，因此需引入 ‌事务型 KV 接口‌，支持多键原子操作及并发读控制。

核心要点

原子性不可组合‌：多个原子 KV 操作（如 set/del）组合后 ‌失去原子性‌（例如更新主数据和二级索引时崩溃，可能导致状态不一致）。

事务接口必要性‌：需 ‌事务型 KV 接口‌ 支持多键原子操作（如原子提交/回滚）和并发读控制。

崩溃恢复要求‌：若操作中途崩溃，系统需回滚到操作前状态（例如通过事务日志实现）。

10.4 关于KV上表和索引的总结

• 行和列作为键值对。

• 范围查询。

– B+树迭代器。

– 保持顺序的编码。

• 二级索引。

– 索引选择。

– 需要事务接口。

**原子事务**‌

11.1 全有或全无效应

上一章节的二级索引要求‌原子多键更新‌。这不仅是‌数据库内部一致性‌的必要条件，也对‌应用层数据一致性‌至关重要（例如账户余额与账户交易的关联场景）。

我们将‌弃用原有的 get-set-del 接口‌，新增支持‌原子化执行多个操作组‌的新接口。关于并发控制的讨论将在下一章节展开。

提交与回滚  
我们将添加接口来标记事务的开始和结束。在事务结束时，更新要么生效（提交），要么因错误或用户请求（中止）而被丢弃（回滚）。

通过写时复制实现原子性  
使用写时复制技术时，提交和回滚操作只需更新根指针。该功能已作为错误处理机制在第06章实现。

此前，单个键更新后会调用 updateOrRevert()。现在该方法已移至 KVTX.Commit() 中。B+树可按需进行多次更新，关键在于根指针的指向。

替代方案：通过日志实现原子性

在写时复制树中，更新通过根指针的指向来捕获（与就地更新方式不同，就地更新需依赖日志记录才能捕获更新内容）。

当日志存在时，若事务被中止，可通过日志回滚已执行的更新。但问题在于：IO错误会阻止后续更新，因此回滚操作需交由恢复机制处理——这一逻辑同样适用于写时复制场景（参见 updateOrRevert 方法）。

一旦日志中记录的更新调用 fsync 完成，这些更新即被视为持久化。因此，只要查询操作会读取日志内容且日志最终会合并到主数据存储中，数据库只需完成一次 fsync 即可向客户端返回操作成功。

11.2 事务接口

将树操作移至事务中

树操作现与事务绑定，因此这些操作已被迁移至 KVTX 中。

请注意，这些函数不再返回错误，因为实际的磁盘更新操作已移至 KVTX.Commit() 方法中完成。

事务性表操作

对于基于表的接口，只需在 KVTX 中新增一个包装器类型，并将这些表操作迁移至该包装器中。

此类操作不再处理 IO 错误，因此更新二级索引时无需进行错误处理。

11.3 可选优化

实现可运行的关系型数据库已是重要里程碑，尽管目前仅支持顺序操作。

若需进一步挑战，可考虑以下优化方向：

减少多键更新时的复制开销‌

写时复制机制在单次更新中会沿着从叶子节点到根节点的路径复制所有相关节点。这种机制对多键更新效率较低，因为事务过程中：中间树中的节点会被分配内存、更新一次后即被删除。

优化思路是：在事务内部，每个节点仅复制一次，并对已复制的节点采用就地更新方式操作。

范围删除

尽管目前已支持多键更新，但删除大量键（例如删除表）仍存在资源消耗问题。删除表的简单方法是逐个迭代并删除键，这会导致整个表被加载到内存中，并在节点被删除前反复更新，造成无效操作。

某些数据库为每个表使用独立文件，因此不存在此问题。而在我们的实现中，所有数据共用一棵 B+ 树，因此可以实现一种范围删除操作：无需访问范围内的叶子节点，直接释放它们。

压缩公共前缀

在有序数据中，相邻的键很可能共享公共前缀。在典型的关系型数据库场景中，多列键也会产生共享前缀。因此，可以在节点内部对键进行压缩。

前缀压缩会增加实现复杂度（但也更有趣），尤其是当节点大小在合并与分裂时不易预测的情况下。

并发控制

12.1 并发层级

问题：读操作与写操作的交错执行‌

多个客户端可以随时进入/退出事务，并在事务间发起读写请求。为简化分析，我们假设"进入事务/退出事务/读/写"都是原子步骤，因此并发事务的执行过程可视为这些原子步骤的交错组合。

区分只读事务与读写事务‌

我们需要区分只读事务（read-only TX）和读写事务（read-write TX），原因在于：

并发读操作的处理难度远低于并发写操作；

许多应用场景是读密集型的，读性能更为关键。

读写锁（RWLock）‌

若缺乏并发控制方案，最简单的方法是使用互斥锁（mutex）将所有数据访问串行化。但为提高读性能，可采用读写锁（RWLock）替代：

无写操作时‌：数据不可变，允许多个读操作并发执行；

有写操作请求时‌：写操作需等待所有读操作完成；

读操作阻塞规则‌：读操作会被写操作阻塞，但不会被其他读操作阻塞。

局限性‌

这种方法的实用性有限，因为：

写操作之间无法并发执行；

长事务会导致性能恶化（读写操作会相互阻塞）。

读-拷贝-更新（RCU）

为避免读操作与写操作相互阻塞，可以让读操作和写操作各自操作数据的独立版本：

读操作‌：通过指针访问不可变数据（视为快照），直接获取当前版本；

写操作‌：单一线程修改自身的数据副本，完成后切换指针指向新副本。

由于采用写时复制（copy-on-write）机制，这种并发级别天然实现。但单一写线程仍存在不足，因为事务的生命周期由客户端控制，可能持续任意长的时间。

乐观并发控制‌

并发写操作会引发冲突，例如：

若事务 TX1 依赖的某个键被事务 TX2 修改，则二者无法同时成功。

需注意，某些看似“仅写”的操作实际隐含读依赖。例如，更新/删除接口会返回键是否被更新/删除，这依赖于该键的先前状态。因此以下场景也属于冲突：

处理冲突的一种方法是：检测到冲突时直接中止事务（TX）。

步骤如下：

事务启动‌：事务开始执行。

读与写分离‌：

读操作基于快照（snapshot）进行；

写操作先在本地缓冲区（buffer）中暂存。

提交前验证‌：在提交前，检查是否存在与已提交事务的冲突。

事务结束‌：

冲突存在‌：中止事务并回滚所有操作；

无冲突‌：将本地缓冲的写操作提交到数据库。

关键点‌：

验证（verify）与提交（commit）是一个原子操作（atomic step）。

这种方法称为‌乐观并发控制‌（Optimistic Concurrency Control），其核心假设是冲突极少发生，因此不主动预防冲突，而是在提交时检测并处理。

补充说明‌：

虽然本文描述的是基于乐观控制的实现，但实际系统中还有其他替代方案（例如悲观锁、多版本控制等）需要了解。

替代方案：悲观并发控制‌

在乐观并发控制中，若发生冲突，事务（TX）将无法继续执行。从应用视角看，这并不理想，因为应用只能通过循环重试来处理冲突。另一种解决冲突的方法是‌通过锁机制预防冲突‌：事务会在其依赖项上获取锁，使得可能产生冲突的事务相互等待。

这种方式看似更优，尤其在之前的例子中（写/删除操作可无阻碍执行）。然而，这仍无法保证事务的最终成功，因为事务可能因‌死锁‌（deadlock）而失败。

死锁的定义‌：当两方各自持有对方所需的（不同）锁，并相互等待对方释放锁时，即发生死锁。这种情形也适用于多方场景——只要依赖图（dependency graph）中存在循环。

死锁的应对‌：

在并发编程中，锁应按预定义的顺序获取以避免循环。

但在数据库中，客户端可以按任意顺序获取锁，因此数据库必须主动‌检测并解决死锁‌——这本质上是一个图遍历问题。

12.2 读操作的事务隔离（Snapshot Isolation）‌

事务隔离级别‌（Isolation level）决定了事务（TX）如何感知其他事务的修改。对于‌写时复制‌（copy-on-write）机制而言，这并非问题，因为事务始终基于 B+ 树的快照进行操作。

本地更新的捕获‌

事务会维护以下两部分：

数据库快照‌：通过写时复制机制生成的 B+ 树根指针（root pointer）指向的静态数据；

本地更新‌：存储在内存中的一个独立的 B+ 树，记录事务执行期间的所有修改。

初始化规则‌：

事务启动时，上述两棵 B+ 树（快照树与本地更新树）均会被初始化。

读取自身未提交的写入‌

在事务（TX）内部，客户端应能读取其已写入但尚未提交的数据。因此，执行查询时，‌应优先检查事务的本地待提交写入（KVTX.pending），再检查数据库快照（KVTX.snapshot）‌。这也是为什么事务的本地写入需存储在 ‌B+树‌ 而非简单列表中的原因——B+树支持高效的范围查询操作。

范围查询的实现‌：

为支持范围查询（range queries），需新增一种迭代器类型（iterator type），用于‌合并事务本地待提交的 B+树‌ 和 ‌数据库快照的 B+树‌，确保查询结果包含事务自身的未提交写入。

空闲列表中的版本号管理‌

由于读操作可能持有数据库的旧版本，因此空闲列表（free list）‌不能释放这些旧版本中的页面‌。解决方案是：为每个版本分配一个‌单调递增的版本号‌（逻辑上称为时间戳）。

具体实现逻辑‌：

事务跟踪‌：系统会跟踪所有进行中的事务（TX）及其所基于的版本号。

页面版本关联‌：每个被添加到空闲列表的页面都会关联一个版本号。

分配限制‌：空闲列表‌绝不会分配版本号比当前最旧事务还新‌的页面。

运作机制‌：

当从空闲列表头部取出页面时，需检查其版本号。

空闲列表采用 ‌FILO（先进后出）‌ 结构，因此‌最早版本的页面会优先被分配使用‌。

修改点1‌：在 KVTX 和 KV 结构中引入版本号。

maxVer 的维护机制‌

当事务（TX）退出时，maxVer 会被更新为 KV.ongoing 中‌最旧的版本号‌。其作用是：在已有的 maxSeq（最大序列号）基础上，‌额外防止页面被提前重用‌。

版本号更新规则‌：

curVar（当前版本号）由‌写入者（writer）‌在提交时设置为‌下一个递增的版本号‌。

12.3 处理写入冲突‌

检测与历史记录的冲突‌

修改点1‌：所有读取操作（包括‌点查询（point queries）‌和‌范围查询（range queries）‌）会被记录到 KVTX.reads 中。

修改点2‌：每次成功的提交操作会被添加到 KV.history（历史记录）中。

冲突检测机制‌：

通过检查事务的‌依赖项‌（即其读取的数据范围）与 KV.history 中‌晚于事务基础版本‌的历史记录是否存在‌重叠‌，判断是否发生冲突。

历史记录的清理‌：

当最旧的事务（TX）退出时，系统会‌清理 KV.history 中过期的历史记录‌，以释放存储资源。

序列化内部数据结构‌

在理论分析中，事务（TXs）被简化为‌交错的步骤‌，但实际场景中这些步骤可能在‌并行线程‌中运行。由于它们共享 KV 结构，因此需要通过锁机制实现‌串行化‌。

锁的使用策略‌

可以为所有 KVTX 方法使用同一把锁，但可通过优化减少锁的粒度。

读/写方法无需串行化‌，原因如下：

写操作‌仅修改 KVTX.pending，不会直接操作 KV。

读操作‌仅访问 KV.mmap.chunks（由 mmap 返回的切片），这些内存映射区域是‌只读的‌。

提交时的数据修改‌

提交操作可能通过‌追加数据‌修改 KV.mmap.chunks。

解决方案：为每个事务（TX）维护一个‌本地副本‌（local copy）。

由于切片（slice）是‌仅追加（append-only）‌的，因此浅拷贝（shallow copy）即可满足需求。

锁机制的优化设计‌

通过上述设计，‌读/写方法无需加锁‌，可以并行执行。这种优化非常重要，因为读操作可能触发‌页面错误（page faults）‌并导致线程阻塞。

当前锁的粒度‌

目前仅对 Begin（事务开始）、Commit（提交）、Abort（终止）操作进行串行化。

但考虑到 Commit 操作涉及 ‌IO 操作‌，可以进一步优化：在等待 IO 时‌释放锁‌，允许其他事务进入或只读事务退出，从而提升并发性。

提交操作的串行化‌

提交步骤仍需通过‌另一把锁‌与其他提交操作保持串行化，以确保数据一致性。

这部分优化细节（如锁的分层设计）‌留作扩展实现时的任务‌。

**SQL 解析器**‌

**SQL 的易解析性与类英语特性**‌  
SQL 既能被计算机轻松解析，同时又保持了类似英语的直观表达形式。

‌**13.1 语法、解析器与解释器**‌  
‌**计算机语言的树状表示**‌  
查询语言（如 SQL）本质上是一个字符串，需通过解析转换为树状结构（如抽象语法树 AST）以进行后续处理。

‌**示例 1：SELECT ... FROM foo WHERE a > b AND a < c**‌

* SQL 只是众多语法中的一种，还存在更简化的替代方案：
  + ‌**PRQL**‌[1]：基于管道的查询语言。
  + ‌**S-表达式**‌[2]：仅通过嵌套括号表达的树状结构语法（最简单且通用的语法形式）。

‌**S-表达式的优势**‌

* 若选择 S-表达式，本章内容可跳过（因其语法极简）。
* 但 SQL 的解析难度并未显著增加，因其完全可通过‌**自顶向下的递归**‌实现处理。

‌**本章知识的普适性**‌

* 本章介绍的解析方法与原理适用于‌**绝大多数计算机语言**‌的解析与处理。

**通过遍历树节点进行求值**‌  
无论是 ‌**SELECT**‌ 还是 ‌**UPDATE**‌ 语句，都可能包含针对列的算术表达式。这些表达式在上一示例中已被解析为树状结构：

* 每个树节点代表一个‌**运算符**‌，其子树则对应‌**操作数**‌。
* 对树节点求值时，需‌**先递归求值其子树**‌（即操作数的结果）。

‌**树形结构的意义**‌

* 树形结构的核心价值在于它天然表达了‌**求值顺序**‌的依赖关系。
* 编程语言虽然包含控制流、变量等复杂概念，但一旦用树形结构表示后，其底层逻辑（如求值步骤）便可通过树遍历自然推导。

13.2 查询语言规范‌

语句结构‌

并非完全遵循 SQL 标准，仅保持近似语法形式。

条件处理机制‌

SQL 数据库的默认行为‌：

若 WHERE 子句的条件可被索引覆盖，则优先‌自动选择索引‌加速查询。

若条件无法完全通过索引满足，则需‌逐行读取数据并过滤‌。

此过程完全由数据库隐式处理，‌用户无法直接干预‌。

与 SQL 的核心差异‌

本设计将条件拆分为两类独立子句，取代 SQL 的单一 WHERE 子句：

INDEXED BY‌：显式指定索引条件（用于快速定位数据）。

FILTER BY‌：显式指定过滤条件（用于逐行校验数据）。

目标‌：赋予用户对查询过程的‌直接控制权‌，而非依赖数据库的隐式优化。

OLTP 工作负载的性能稳定性要求‌

OLTP（联机事务处理）场景的核心诉求‌：需保障‌可预测的性能表现‌。

潜在风险‌：若数据库‌查询计划突然变更‌（如优化器选择不同索引），可能引发生产环境事故。

设计原则‌

显式索引选择‌：通过语法强制用户‌明确指定索引‌（而非依赖数据库优化器隐式决策），从而‌避免数据库自行猜测索引带来的不确定性风险‌。

表达式结构‌

表达式可定义为以下任一形式：

列名‌（例如 user\_id），

字面量值‌（如数字 123、字符串 "hello"），

二元或一元运算符‌（如 +, -, NOT），

元组‌（例如 (1, 2)）。

这些元素均以‌树节点形式‌在语法树中表示。

运算符优先级处理‌

不同运算符具有不同的‌优先级‌（例如乘除优先于加减），优先级规则已在前文列出。

更简单的语法（例如 S 表达式）可规避此复杂性，但运算符优先级问题可通过‌简单的递归方法‌解决（后续示例将展示具体实现逻辑）。

13.3 递归下降解析与树节点结构‌

树节点结构定义‌

每条语句均被拆解为更小的组成部分（包括表达式节点 QLNode），形成由组件构成的‌树形结构‌。

解析过程逻辑‌

输入拆分‌：将输入语句‌逐层分解为更小的语法单元‌。

自上而下解析‌：

解析器首先识别最顶层的‌语句类型‌（如 SELECT, INSERT）。

根据语句类型，将解析任务‌分派到具体的处理函数‌（如 parse\_select(), parse\_insert()）。

递归展开‌：每个处理函数进一步递归解析子节点（如表达式、条件子句等），最终构建完整的语法树。

解析核心逻辑‌

即使不深入每个函数细节，解析的核心思路已然清晰：

输入逐层分解‌

递归拆分过程‌：将输入语句‌逐层分解为更小的语法单元‌，直至无法再分解（最终单元可能是运算符、标识符名称或字面量值）。

基于关键词的路由机制‌

动态判断解析路径‌：通过‌前瞻下一个关键词‌（如 WHERE, JOIN）决定后续解析逻辑（如跳转到条件解析或表连接解析）。

将中缀运算符转换为二叉树‌

表达式解析函数 pExprOr 负责解析任意表达式。将 1+2\*3-4 这类包含运算符优先级的表达式转换为二叉树结构并不直观（因其本质是运算符与操作数的交错序列）。因此，我们‌从简化问题入手‌：

阶段一：仅处理加法运算符‌

即使表达式包含多个加法运算（如 1+2+3），仍可将其转换为‌二叉树结构‌。

伪代码示例‌：

python

Copy Code

def parse\_additive\_expression():

left = parse\_primary() # 解析基础项（如数字、列名）

while 下一个运算符是 "+" 或 "-":

op = 读取运算符

right = parse\_primary()

left = BinaryOpNode(op, left, right) # 构建二叉树节点

return left

阶段二：扩展至多级优先级运算符‌

通过‌递归分层解析‌（如先解析乘除，再解析加减），可逐步处理复杂优先级规则，最终生成完整的表达式树。

核心规则描述‌

该逻辑遵循以下规则：

expr := expr + term

expr := term

递归结构‌：左侧子规则 expr ‌可递归展开为自身‌（如表达式嵌套包含自身），而右侧子规则 term 是‌无法再分解的基础单元‌。

运算符优先级的递归处理‌

新增运算符‌：引入更高优先级的 \* 运算符后，需调整规则结构。

层级细化‌：

原基础单元 term 被扩展为类似规则：term := term \* factor

此时，factor 成为新的‌最底层不可分解单元‌（如字面量、列名或括号表达式）。

优先级实现‌：通过递归层级（expr → term → factor）‌隐式定义优先级‌，深层规则对应更高优先级运算符。

运算符优先级与递归下降解析‌

OR运算符优先级最低‌：因此，解析表达式的顶层入口函数是 pExprOr。

逐层向下解析‌：

pExprOr 调用 pExprAnd 处理‌下一级优先级‌（如AND运算符）。

此过程‌递归向下‌，直至最高优先级的 pExprUnop（如一元运算符 - 或 NOT）。

最终，pExprUnop 调用 pExprAtom 解析‌最底层不可分割单元‌（如标识符名称 id 或字面值 42）。

本质：递归下降法‌

分治策略‌：将复杂表达式拆解为优先级逐层升高的子问题（“分”的过程体现为‌检查下一个关键词‌）。

自顶向下构建语法树‌：通过函数调用链隐式实现运算符优先级，最终生成完整的表达式树结构。

查询语言‌

14.1 表达式求值‌

SELECT 与 UPDATE 的表达式求值‌：

这两个语句中涉及的‌列（columns）上的表达式‌需进行求值（如 salary \* 1.1）。

表达式树的求值逻辑‌：

其过程‌与前一章讨论的树结构求值方法一致‌（通过递归遍历二叉树计算结果）。

INSERT 的特殊处理‌：

INSERT 语句中的‌常量表达式‌（如 NOW() 或 2 + 3）需在‌空环境（empty env）‌下求值（即无数据行上下文参与计算）。

14.2 范围查询‌

范围查询的构建‌

适用范围‌：

SELECT、UPDATE 和 DELETE‌ 语句均可执行范围查询，其‌区别仅在于对查询结果的操作‌（如 SELECT 返回结果，UPDATE 修改结果，DELETE 删除结果）。

公共组件 QLScan‌：

QLScan‌ 是表示范围查询的‌公共逻辑模块‌，负责统一处理查询范围的定义。

QLScan 的三阶段处理流程‌

INDEX BY（索引阶段）‌：

由 ‌Scanner‌ 组件实现，用于‌基于索引快速定位数据范围‌（如利用 B+ 树索引加速查询）。

LIMIT（限制阶段）‌：

对索引扫描结果‌施加行数限制‌（如 LIMIT 100）。

FILTER（过滤阶段）‌：

对受限范围内的数据‌应用过滤条件‌（如 WHERE salary > 5000）。

重新审视无限编码‌

INDEX BY 的三种形式‌（基于前一章定义）：

区间查询‌：a > start AND a < end

表示开区间 (start, end)。

开区间右无限‌：a > s

表示右无限开区间 (𝑠, +∞)。

索引前缀匹配‌：a = p

表示基于索引前缀的精确匹配（如索引为 (𝑎, 𝑏)，则仅匹配 a = p 的行）。

索引前缀的处理逻辑‌

精确匹配等价性‌：

a = p 等价于 a >= p AND a <= p，其编码形式为：

(𝑎, 𝑏) ≥ (𝑝, −∞) 且 (𝑎, 𝑏) ≤ (𝑝, +∞)。

右无限区间的分解‌：

a > s 等价于 a > s AND () <= ()，其编码形式为：

(𝑎, 𝑏) > (𝑠, −∞) 且 (𝑎, ) < (+∞, )（空元组 () 表示无后续条件）。

关键调整‌

空元组 () 的影响‌：

由于空元组的引入，索引键 Key1 和 Key2 的列组合可能不同（例如 Key1 包含部分列，Key2 包含全列）。

需‌修改索引选择逻辑‌以支持此类混合列组合的编码方式。

14.4 总结与下一步计划‌

持久化事务型数据库的多接口支持‌

我们为数据库提供了多种访问接口：

键值存储（KV）‌：可‌嵌入应用程序‌直接使用。

关系型数据库‌：同样支持‌嵌入式部署‌。

类 SQL 查询语言‌：为关系型数据库提供‌声明式查询能力‌。

扩展方向‌

网络协议支持‌：

无需新增功能，即可通过‌网络协议‌实现数据库的‌跨进程或跨机器调用‌。

Go 语言的网络编程抽象层级高且易用，但若想‌从零实现网络协议‌（如深入理解底层细节），可参考《构建你的 Redis》[1]一书。

编译器的探索‌：

当前已具备基础的‌解析器（Parser）‌和‌解释器（Interpreter）‌，下一步可尝试开发编译器（Compiler），将自定义编程语言‌编译为机器码‌（而非仅解释执行）。

相关技术可参考《从源码到机器码》[2]一书。